8-14-2018

Reliable Physical Unclonable Function Design and Algorithm for Authentication and Key Generation

Wei Yan
University of Connecticut - Storrs, wei.2.yan@uconn.edu

Reliable Physical Unclonable Function Design and Algorithms for Authentication and Key Generation

Wei Yan, Ph.D.

University of Connecticut, 2018

Security is becoming an increasing concern in electronic devices recently. Specifically, since the embedded systems and Internet of Things (IoTs) have become necessary parts of our life, more and more vulnerabilities are detected and made use of by attackers. Moreover, as the electronic component supply chain grows more complex due to globalization, with parts coming from a diverse set of suppliers, counterfeit electronics have been a major challenge that calls for immediate solutions. This is because the traditional solutions that using static digital ID and keys can be easily obtained or cloned. The current best practice is to place a secret key in non-volatile memory such as fuses and EEPROM, and use cryptographic primitives to authenticate a device and protect confidential information. To reduce the vulnerability of the systems, we have developed multiple methodologies in this work. The proposed methods include: the design optimization and implementation of ring oscillator physical unclonable function (RO PUF) on field programmable gate arrays (FPGA); a novel phase calibrated RO PUF and

the corresponding authentication solution; a PUF initialization table (PIT) that provides high accurate authentication; a PIT-based floating thresholding algorithm for key generation; a lightweight ring weight algorithm (RWA) that can be applied to the low-cost authentication; an efficient locality sensitive hash function (LSH) for not only similarity search, but also data clustering.

Reliable Physical Unclonable Function Design and Algorithms for Authentication and Key Generation

Wei Yan

B.S., Nanjing University of Information Science and Technology, 2011

M.S., Suzhou Institute of Nano-Tech and Nano-Bionics, Chinese Academy of Science, 2014

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2018

APPROVAL PAGE

Doctor of Philosophy Dissertation

Reliable Physical Unclonable Function Design and Algorithms for Authentication and Key Generation

Presented by

Wei Yan, M.S.

Major Advisor

John A. Chandy

Associate Advisor

Omer Khan

Associate Advisor

Lei Wang

University of Connecticut

2018

Dedicated to my parents.

ACKNOWLEDGEMENTS

First and foremost, I have to sincerely thank my supervisor, Dr. John Chandy, for his great guidance and support. His steadfast support of my Ph.D study and research was greatly needed and deeply appreciated. His sage advice, insightful criticisms, and patient encouragement aided the writing of this thesis in innumerable ways.

Besides my advisor, I would like to thank the rest of my thesis committee: Dr. Lei Wang, Dr. Omer Khan, Dr. Zhijie Shi, and Dr. Faquir Jain for their insightful comments and encouragement.

Furthermore, it is great honor to work with my friends, Fatemeh Tehranipoor and Chenglu Jin, who have always been a constant source of support and encouragement during the challenges of my Ph.D years. To all my friends, thank you for your understanding and encouragement in many, many moments of crisis. Your friendship makes my life a wonderful experience. I cannot list all the names here, but you are always on my mind and in my heart.

Last but not the least, I want to thank my parents, who have taught me to work hard for the things that I aspire to achieve.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

In recent times, electronic devices have become so popular in our life, making people rely on the safety and security of those embedded systems and IoTs. When we store our sensitive information on these untrusted devices, the system vulnerabilities offer the attackers great opportunities of obtaining individual information. Unfortunately, though many conventional solutions have been applied to improve the security level, there are always new adversaries developed. As a fundamental reason, the vulnerabilities come from the hardware design and supply chain. To solve the problem, physical unclonable functions (PUF) are recommended as a secure solution of authentication, random number generation and key generation, which provides multiple features against hardware and software attacks. By implementing PUFs in the systems, commercial products overcome the traditional security shortages to a certain degree.

1.1 Security Vulnerabilities of Embedded Systems and IoTs

Not as much work has been done in secure system design despite its importance in establishing security and trust for embedded systems, SoCs, and IoTs. The vulnerabilities of these systems include physical tampering, malware attack, information leakage, and other aspects. The main threat of hardware is the counterfeit chips in the supply chain, which contains processors (ASICs, FPGAs, and microprocessors), nonvolatile memories (Flash memories, ROM, and RAM), IPs, and PCBs. As a consequence, effective protocols are required to authenticate and authorize each component in the system during the initialization. Apart from countermeasures, a secure hardware should also consider some methods against the probing attacks, snooping attacks, hardware Trojans, power analysis attacks, electromagnetic attacks and machine learning attacks. On the software side, there are brute-force attacks, man-in-the-middle attacks, buffer overflow attacks, fault injection attacks and others. To protect the system, people developed secure programming languages with trusted compiler and libraries, architecture and OS based countermeasures, static and dynamic code analyzers, and sandboxing approaches. Nevertheless, a system level design flow with security consideration has not been discussed widely yet.

1.2 Hardware Security Components

As we know, the traditional security solution mainly relies on the cryptography algorithms. Though they are very practical in reality, researchers have successfully devel-

oped side-channel attacks against many hardware cryptography implementations. Instead, trusted platform module has been proposed as a complex security solution in the computer system. However, such a high-cost design cannot be applied to lightweight embedded systems or IoTs. Meanwhile, different true number random generators (TRNGs) have been proposed these years. Hardware RNG can be used for key generation in certain applications, but it does not provide deterministic results. Thus, the application is limited. Another security related module is one-way hash function. Though it is deterministic and stable, complex computation is required.

To overcome the limitation of those existing methods, researchers tried to use the random process variations that are generated during the manufacturing process. Since the components and circuits on the board have their own uncontrollable parameters, it is possible to use the uniqueness to create unpredictable response. With this consideration in mind, physical unclonable functions were proposed by Gassend et al [1]. As a physical component, a PUF is very hard to duplicate or predict due to the results of random uncontrollable variables in the manufacturing process. When a PUF receives a challenge, it reacts with a response, which is known as a Challenge-Response Pair (CRP). An ideal PUF-based CRP provides strong advantages in that each response gives negligible information on the responses from different challenges to the same PUF or even identical challenges on different PUFs [2]. Rather than storing secrets in non-volatile memories, PUFs can provide significantly higher physical security by generating these secrets from unique PUF responses [3]. A PUF can be broadly classified as a “strong

PUF" or a "weak PUF". The fundamental difference between weak and strong PUFs is the number of unique challenges that the PUF can process [4]. Both contain the same characteristics including randomness, uniqueness, and reliability. A PUF provides random output according to different input by involving the physical variations in the manufacturing process. This is significantly important in random number and secure key generation. Secondly, the output of PUFs must be unique. Given the same input, two PUFs output different values. As a result, it is hard to predict the responses of one PUF even if all the data is tested on another PUF. Finally, an ideal PUF should generate deterministic output, which is the basic requirement of key generation and authentication applications. Therefore, PUFs can be applied as a source of random but reliable data for applications such as generating unique IC identification numbers or encryption keys.

1.3 Problem Statement

Though PUFs have many advantages, it is not a perfect primitive for security applications. The biggest challenge of using PUFs is its instability in the output. It is essential that the PUF responds with consistent results given the same challenge. However, most PUF constructions are not 100% reliable. Because the physical variations have very limited effect on the output, the result sometimes stays in a metastability condition. Meanwhile, temperature, voltage, and other factors affect the stability to some degree. For TRNGs, bit errors may be positive and acceptable. However, authentication and

key generation applications require deterministic values in most scenarios. The high raw bit error rate (RBER) in the responses of PUFs needs strong error correcting codes (ECC) to guarantee a low false negative rate. However, a strong ECC requires large memory space and high calculation complexity. When the decoding demands are large enough, electronics systems with limited computation and storage capabilities may not be able to afford the burden. As a result, traditional PUFs may not offer enough reliability for industrial applications. In this work, we will mainly discuss how we solve the instability of PUFs. Multiply solutions are offered to address the problem under different applications and requirements.

1.4 Related Works and Limitation

To solve the instability issue of PUFs, post-processing methods are widely involved. Since security-critical products require very low bit error rate of their cryptographic keys, those optimized PUFs cannot be used directly as key generators. Therefore, various HDAs have been proposed to maximize the stability in PUF responses. As industrial practical solutions, error correction schemes ensures the key to be reproducible [5]. For example, Bose-Chaudhuri-Hocquenghem (BCH) codes can provide high accuracy by encoding and decoding data with large arrays [6, 7]. However, ECC also means heavy redundancies. When the key length grows, the required codeword can be extremely large while the decoding time becomes significantly long. This redundant data also leaks information about the PUF response itself leading to a potential security

vulnerability. Another disadvantage is that ECC has a strict limitation on the maximum number of bits it can correct. Given a number of errors beyond its capability, ECC will lose its functionality. Alternatively, bit selection schemes become popular recently [5, 8, 9]. As a lightweight soft-decision coding, the least reliable bits in responses are discarded during the bit selection process. The most intuitive idea is to impose a global threshold and filter out all mismatched bits. Relatively, local thresholding retains only the most reliable bits [5]. However, those algorithms have poor resistance to the variety of models caused by process variation or environmental noise. The reason is that some unstable bits will either cross the threshold or change the sequence, which means former models are no longer suitable. As a result, it is hard to regenerate the same bit set with the original sequence, and no bit errors cannot be guaranteed. According to the HDAs described above, both error correction and bit selection schemes are not satisfying in term of PUF stability improvement.

1.5 Contributions

In this work, we devote the main paragraph to the design and algorithm of PUF stability, which includes the optimization design, implementation, various authentication solution, and key generation method. Specially, the contributions are list as follows:

1.5.1 PUF Design Optimization and Implementation

We optimize the traditional PUF design by replacing the challenge register with a shift register or hash function. We also modify the architecture for timing improvement. Details of implementing PUF on FPGAs are discussed. Furthermore, a phase calibrated PUF is proposed to reduce the bit errors.

1.5.2 Post Processing Authentication Algorithm

We propose several authentication algorithms for different scenarios. Ring weight algorithm is a lightweight solution with limited computation, overhead, and accuracy. PUF initialization table provides very low false negative and false positive, though the calculation amount is much larger than RWA. Unstable-stable response is a trade-off between those two methods.

1.5.3 Solution for Key Generation Applications

By recording the feature of PUF in PITs, a floating thresholding key generation algorithm is discussed in this work. We involve both bit selection and error correction methods in the solution to improve the stability as much as we can. Meanwhile, security problems are also considered since it is a critical factor of the application.

1.6 Thesis Outline

- Instruction

- Background
- PUF Design Optimization and Implementation
- Secure Authentication Solution with PUF Response Instability
- PUF Initialization Table for Robust Authentication and Key Generation
- Ring Weight Algorithm for Lightweight Authentication
- Locality Sensitive Hash Function for Identification and Clustering
- Conclusion

Chapter 2

Background

Physical Unclonable functions map a set of inputs (challenges) to a set of outputs (responses) which are called challenge-response pairs (CRPs). During the PUF design and implementation, we make use of random variations of manufactured structures to derive an uniquely individual behavior for each Integrated Circuit (IC). In order to harness theses PUF properties, these individual variations must be extracted and mapped; which is almost impossible since process variations and noises are not clonable. PUFs came to the stage at a time when traditional cryptography failed to stand its ground against physical attacks, side-channel attacks, etc. A significant advantage of PUF-based authentication is that unlike traditional key-based cryptographic systems, a PUF does not require the storage of secret keys in non-volatile memory. Instead, the secret key is hidden within the physical body (e.g. intrinsic properties) of the circuitry which is found to be unclonable and unpredictable. In this chapter, we will introduce several popular PUFs.

2.1 Strong and Weak PUFs

PUFs are usually classified into strong and weak depending on the number of challenge-response pairs (CRP) that they can handle. Weak PUFs support a small number of CRPs which are linearly related to the number of components used to build the PUF. They possess essentially a single, fixed challenge, as for example in the case of SRAM PUFs and DRAM PUFs. Strong PUFs, on the other hand, support a large number of CRPs such that polynomial time attacks become unfeasible. The lack of access restriction mechanisms on strong PUFs is therefore a key difference from weak PUFs. These type of PUFs are usually employed for device authentication applications. As with any embedded system design and IoT devices, there is an inherent trade-off between the constraints of the operational environment and the desired functional capabilities of the device.

2.2 Ring Oscillator PUF

A ring oscillator PUF (RO PUF) is a typical example of delay PUFs whose instability has been heavily studied but remains unsolved. Figure 2.1 shows the traditional RO PUF design, which contains an array of ring oscillators (RO), two multiplexers (MUXes), two counters, and a comparator. Because of manufacturing variations, the wire delay of each RO is not controllable, which leads to different frequencies of the RO output. By selecting two ROs according to the PUF input, we can measure the pulses in a defined unit time with the counters. For example, if the first counter holds

the larger value than the second one, the PUF output is '1'. Otherwise the PUF output is '0'. Ideally, the output keeps the same bit value giving a certain input, but in reality bit errors and bias are involved in the output. Though the systematic or correlated process variation and the environmental noise caused by the voltage and temperature variations degrade the output stability [10], the bit errors of FPGA-based PUFs are directly generated by a selected pair of ROs with close frequencies, which lead to the unstable measurement in the counters and the flipped output in the comparator. In [11], RO PUFs are characterized over 125 FPGAs. To improve the quality of ROs, the surrounding logic effect on the oscillator frequencies was studied and a strategy was proposed by placing and comparing ROs in a chain-like structure [12]. A reliability-improvement technique was used in pre-quantization phase of RO PUFs to reduce the noise in PUF responses [13]. More optimized approaches were mentioned in [14] by introducing configurable ROs. They also suggested to compare adjacent RO pairs by controlled RO placement, but the FPGA implementations were not clear. A group-based RO PUF was introduced in [15], which described a framework to filter the systematic variation and improve the hardware efficiency. However, its responses still required ECC.

2.3 Arbiter PUF

The arbiter PUF (APUF) is another typical delay PUF, which is widely studied and evaluated [16]. An Arbiter PUF extracts the device-specific variation as delay difference between two selector chains. In another word, the generic idea behind arbiter

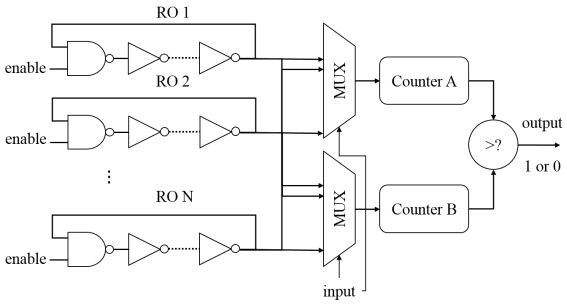

Fig. 2.1: The architecture of a traditional RO PUF

PUF is to race the delay times between two signals. The two signals propagate through various routes depending on the value of the challenge. As shown in Figure 2.2, each challenge bit selects the input path of a MUX pair. Since the location of the MUXes are different, the input wire delay may not be exactly the same. The final value of the 1-bit response is determined by which signal arrives first. Thus, its output should be either '0' or '1', but it is unpredictable. Note that the pattern is also determined by the input length. If the challenge has n bits, the propagation delay time should have 2^n combinations. Early work on Arbiter PUF modeling attacks had described successful approaches already, including the standard Arbiter PUF, XOR Arbiter PUF, lightweight PUF, and Feed-Forward Arbiter PUF [17, 18].

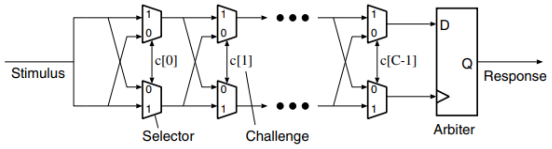

Fig. 2.2: The architecture of a traditional Arbiter PUF

2.4 Bistable RO PUF

Similar to an SRAM cell that contains a pair of cross-coupled inverters, the bistable RO PUF (BR PUF) is based on the fact that an inverter ring consisting of an even number of inverters has two possible stable states. Figure 2.3 shows an example of a 64-stage BR PUF [19]. Each stage contains two NOR gates, a DEMUX as gate input, and a MUX as gate output. Thus, when a 64-bit challenge is received by this BR PUF, either the upper or the lower paths of the MUX and DEMUX are selected. When we apply different challenges, different combinations of NOR gates and paths are selected as a result. In Figure 2.3, the total number of input patterns is 2^{64}. In order to generate a response, the whole PUF needs to be reset as "high" first. After the challenge is applied to the bistable rings, we wait until the bistable rings to be '0'. A reset of "low" is launched afterwards. Then we wait for some time to let the bistable ring stabilize. Finally, we read out a 1-bit response. To generate more bits for responses, we repeat the progress for certain loops.

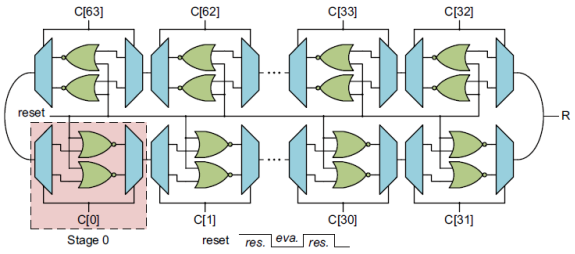

Fig. 2.3: The architecture of a typical BR PUF

2.5 Butterfly PUF

A Butterfly PUF (BPUF) includes a cross-coupled circuit, which is brought to the unstable state until it becomes stable. To make it clear, a cross-coupled circuit provides a positive feedback to record the value in a loop. It has two stable state and one unstable state. The unstable state can be easily changed to one of the stable states by a certain input. The behavior of BPUF is similar to the SRAM cell. As Figure 2.4 shown, the structure of 1-bit BPUF is as simple as two latches [20]. When the input signal is set to high, the BPUF becomes unstable at the beginning. If the input is set to low then, the BPUF turns to be one of the stable state, with an output of '0' or '1'. The final state only depends on the wire delay of the circuit. Originally, since attackers cannot measure the internal delay of an FPGA or ASIC directly, it is hard to predict which state the BPUF is in. However, as the side-channel attacks become popular, they brings new vulnerabilities to BPUFs.

2.6 DRAM PUF

DRAM PUF (DPUF) was recently proposed as a new type of PUF [21, 22], compared to the widely studies of SRAM PUF [8, 23]. According to Figure 2.5, there are three approaches to use DRAM as PUFs. The first one makes use of the built-in-self-refresh module of DRAM chips. As we know, DRAM not only requires a power supply to retain data, but must also be periodically refreshed to prevent their data contents from fading away from the capacitors in their integrated circuits. Therefore, we can initialize

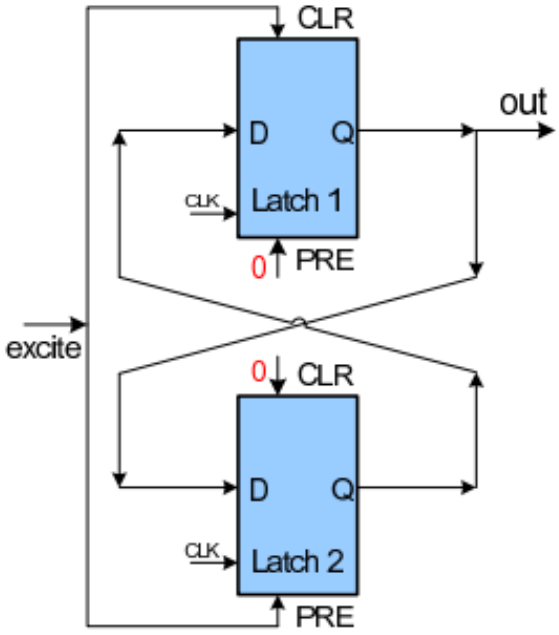

Fig. 2.4: The architecture of a typical butterfly PUF

all cells to '1' and then turn off with refresh. Some cells will leak to '0' while others are still '1'. Another approach is to use the remanence of DRAM cells. When a DRAM powers off, the data fades away over time in a random way. Thus, the remanence effect can also be applied to PUF. The final method is to use the startup value of DRAM. During startup, the storage capacitor has neither been charged nor discharged. Thus the voltage of the capacitor is equal to the bias voltage. Because of the manufacturing variations, the capacitance will be slight different, which means a random value assignment.

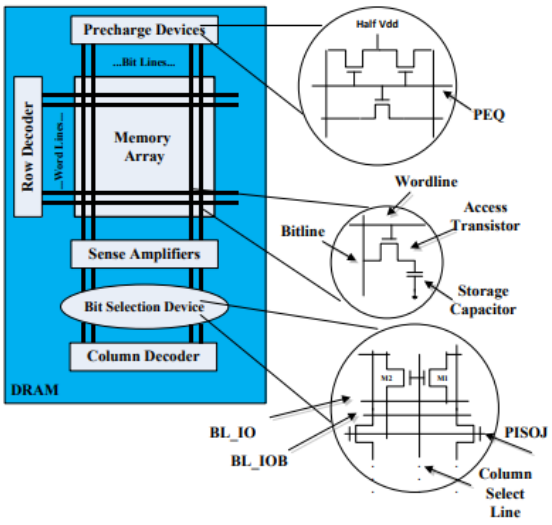

Fig. 2.5: The meory structure of a DRAM array

2.7 Performance Comparison

Table 2.1 shows different types of implemented, and designed, PUFs performance comparison in details. As shown in this Table, the first column demonstrates various performance metrics that need to be considered for each PUF in order to evaluate the effectiveness of each design. Note that the data of RO, abiter, BR, and butterfly PUF are from our implementation and measurement. We also list the memory-based PUF as a comparison. One should notice that the other major difference between the first four presented PUF designs and the fifth (DRAM) PUF is that the "*PUF Type*" differs between strong (large number of CRPs) and weak (small number of CRPs). The next metric that we evaluated was "*Key length*", which was "128 bits" for every one of the PUF designs we examined. The "*lookup table*" and "*flip flop*" rows show the number of those elements used to implement each PUF solution on an FPGA. One should note that

DRAM has no values for this, since it is implemented using a commercial off the shelf (COTS) DRAM memory and not an FPGA. The "*basic cells*" row show the number of cells and type of cells (e.g. multiplexers, ring oscillators, inverters, demultiplexers, flash, memory) that we required to implement each PUF solution. The next metric we evaluated each PUF design using was "*bit error rate*"; defined as percentage of bit flips that occur. One should note that this is a good representation of the stability of each type of PUF. The "*bias*" represents the percentage of 1s and 0s in the generated output. Please note that the presented bias is in terms of the number of 0s present in the output. The following metric, "*uniqueness*", is a measure of how uncorrelated the response bits are across multiple implementations of the same PUF design. Ideally the response bit should differ with a probability of 50%. A key point to make on the uniqueness presented is that the RO and DRAM PUFs have a difference of half the bits (e.g. 50%). The last property is "*randomness*"; a measure of the unpredictability of the response bits. For our proposed PUF, the randomness results show promising capability of our new models. From this table of data we are able to compare and contrast the various solutions to determine those most favorable to any given scenario.

2.8 Contributions

In this chapter, we introduce five types of PUFs: Ring Oscillator PUF, Arbiter PUF, Bi-Stable PUF, Butterfly PUF, and DRAM PUF. The details of design and implementation are discussed in each section. Finally, we compare the performance of these PUFs. The

Table 2.1: Different PUF Performance Comparison

PUF Design	Arbiter	RO	BR	Butterfly	DRAM
PUF Category	Delay-based	Delay-based	Delay-based	Delay-based	Memory-based
PUF Type	Strong	Strong	Strong	Strong	Weak
Key Length (bit)	128	128	128	128	128
Lookup Table	97	3316	75	38960	N/A
Flip Flop	68	2675	30	16912	N/A
Basic Cells	56 MUXes	256 ROs	28 DE/MUXes	32768 Latches	921 DRAM Memory
Bit Error Rate	3.31%	7.12%	1.42%	6.80%	1.6%
Bias	-9% to 8%	-5% to 5%	-8% to 6%	-11% to 9%	-5% to 5%
Uniqueness	≈ 59/128	≈ 64/128	≈ 51/128	≈ 44/128	≈ 64/128
Randomness	> 92%	> 98%	> 94%	> 89%	> 98%

hardware cost, bias, uniqueness, randomness, and stability are evaluated.

Chapter 3

PUF Design Optimization and Implementation

3.1 Introduction

Researchers have developed different types of PUFs since the concept was proposed [11, 16, 19, 20, 22]. Among those PUFs, RO PUF is a well-known type with typical performance. Its architecture can be easily implemented on digital circuit. Therefore, in this chapter, we focus on a practical RO PUF implementation on FPGAs. We optimize the architecture of RO PUFs and show the details of implementing RO PUFs on FPGAs. To improve the measurement accuracy of RO frequency, we use phase calibration process for frequency estimation. Meanwhile, We also minimize hardware resources and develop a flexible output data width. Furthermore, we explore the feature of instability in PUF responses and provide a new design that outputs stable bits according to the unstable RO pairs.

3.2 RO PUF Description and Properties

Problem Analysis

Few papers have discussed the impact of bias in RO PUF responses [14], which is the favoring of RO PUFs to either 1 or 0. According to our tests, we found that the counter design in the traditional RO PUF can contribute significantly to this bias. When *CounterA* and *CounterB* hold the same value, the comparator must make an choice of how to treat this case - in our example, the comparator puts this "equivalent" case in the "smaller" group.. The "larger" group, on the other hand, can be active only when *CounterA* receives more pulses than *CounterB* in the given clock cycles. Therefore, the output contains more 0's than 1's. If ROs are placed more regularly, the frequencies may end up being closer, thus causing more "equivalent" cases. This is aggravated especially when the counter size is limited. Regardless of the count cycles, the bias caused by the counter design is still as high as 5.03%. As a result, the bit error rate and the bias are intensified by the design fault and inaccurate measurement. Another problem of this RO PUF design is the inefficient usage of hardware resources. To generate just a single bit output, the design requires an log N-bit input and N oscillators. Though we can apply a hash function to generate different challenges, its hardware implementation requires even more FPGA slices than the PUF itself [24]. A transient effect ring oscillator PUF was proposed with a good ratio of PUF response variability to response length, but the intra-device variation increases to 1.7% [25], compared to [11]

with 0.86% intra-device variation. In this chapter, we will discuss a practical phase calibration technique to improve the quality of RO pairs comparison, which shows a trade-off between the cost, stability, and flexibility.

3.2.1 RO PUF Design Improvement

Though RO PUFs have been heavily studied, there are still possibilities of improvement particularly with respect to implementations on FPGAs. In this section, we will discuss some practical RO PUF optimizations on the hardware.

Ring Oscillator Design on FPGAs

While PUFs have been primarily targeted towards ASIC designs, we explore their potential on FPGAs. According to our test on the Kintex-7 FPGA [26], the frequency of a 5-stage inverter chain RO is approximately 475 MHz when the system clock is 200 MHz. In the crossing timing domain between the ROs and PUF control logic circuit, a high RO frequency adds to the instability of measurements. As with all oscillators, the rate of oscillation is determined by the length of a delay implemented in a loop. Thus, to reduce the frequency, more inverters can be added in the ROs, but it requires more hardware resources. We provide an improved RO design that takes advantage of the lookup tables (LUTs) of the configurable logic blocks (CLBs) and the general purpose interconnect [27] in the FPGA. As shown in Figure 3.1, the first part consists of a 4-LUT delay and a 1-LUT inverter. The reset signal is used to control the enable timing

of all LUTs. In the second part, a single inverter in the loop implements a high gain inverting amplifier. The output frequency is divided by 2 in order to eliminate output glitches. This design only needs six LUTs and a D-type flip-flop (FD).

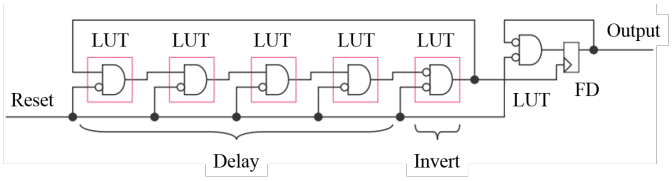

Fig. 3.1: An improved RO design for an FPGA

Shift Register and Hash Function for Resource Improvement

In order to generate variable length responses, we choose a design as shown in Figure 3.2. This example uses 128 ROs to provide 128-bit CRPs. Once a challenge is received, it is stored in a shift register. We select the first 7 bits from the register as the input of the upper decoder and MUX in order to select an RO from the array. Likewise, another RO is selected using the next 7 bits from the challenge shift register. If the addresses are the same, the second 7-bit address is added by 1 to avoid selecting the same RO for comparison. Next, we shift the challenge register to select a new RO pair. The shift pattern can be complex for security consideration. To make it easy to understand, we shift one bit to the left each time. If the challenge is 128 bits, we can generate a 128-bit response with 127 shift operations.

Alternatively, the shift register can be replaced by a non-cryptographic hash function. Compared to cryptographic hash function, which provides one-way, collision re-

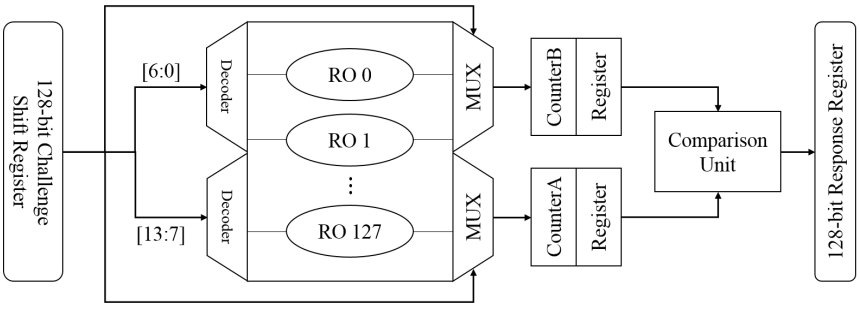

Fig. 3.2: Architecture of our RO PUF

sistant, and deterministic properties, non-cryptographic hash function is significantly faster and low-cost. It is also easy to be implemented on FPGAs. Below is the algorithm of ElfHash that we used in the PUF:

Timing Improvement

In the traditional design, metastability can exist at the input of the counters. It occurs when the counters receive a clock rising edge for comparison. If the RO output happens to flip when a counter value is changing, the result of comparison will be none of larger, smaller, or equal, but a metastable state. We have verified this scenario on our FPGA implementations. As a result, the instability of responses increases. Our solution is to use two registers to store the counter values. The registers and counters are in the same clock domain so that metastabilities are eliminated. The comparison result is recorded in the response register as an output bit.

Algorithm 1 ElfHash process

```
1:  procedure PCP
2:      hash ← 0
3:      x ← 0
4:      count ← 0
5:  loop 1:
6:      hash ← (h << 4) + challenge
7:      challenge ← challenge + 1
8:      if x = hash and 0xF0000000 then
9:          hash ← hash xor (x >> 24)
10:     hash ← hash and (not x)
11:     count ← count + 1
12:     if count = 128 then goto loop 1.
```

3.2.2 RO PUF Implementation on an FPGA

Few papers have clear descriptions on how the RO PUF is implemented on FPGAs [14, 28, 29]. Incorrect HDL coding can lead to logic unit rebuilding during synthesis, which results in functionality failure. For example, inverters built with `inv(1) <= not inv(0);` statements will be optimized out and cannot be placed as RO arrays.

LUT-based RO Placement

To implement the ring oscillators on FPGAs, we use the LUTs directly in our code and add element placement constraints to fix the LUTs with certain delay. In our work, we used the Vivado development kit and the Kintex-7 FPGA provided by Xilinx to show the RO PUF implementation on FPGAs [26, 30]. In the Kintex-7, a CLB contains a pair of slices, and each slice is composed of four 6-input LUTs and other elements [31]. By setting the initial value to "01", a 1-input LUT (LUT1) can perform the same function as a digital inverter. An example of the traditional RO implementation is shown in Figure 3.3. We place one LUT3 (3-input LUT) in the left slice and four LUT1's in the right slice. These five LUTs and the white connections form a basic RO. Apart from the feedback input of another inverter, LUT3 also includes a reset input and an enable input. The left LUTs in a CLB are reserved for other logic of the RO.

Figure 3.4 shows the RO placement given by Figure 3.1. Four delay LUT2s are placed in the left slice and the invert LUT locates in the right slice. The output of the first stage oscillator is connected to the FD, which builds another oscillator with another

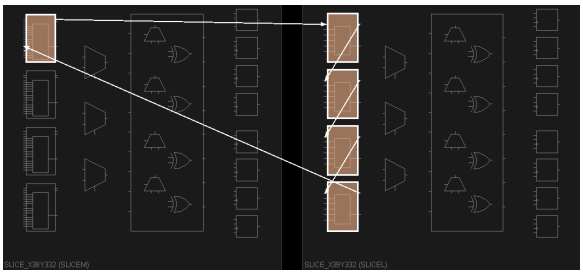

Fig. 3.3: LUT-based RO placement example

LUT2 in the right slice. Since the delay LUTs contain no logic, 'keep' attributes is required in the design to stop logic optimization by the synthesis tool. Additionally, a combinatorial loop in the FPGA is considered bad design practice in most cases, which increases the number of cycles by infinitely going around the circle in the same path. To avoid unnecessary errors during synthesis and bitstream generation, some constraints are applied to let Vivado ignore these loops.

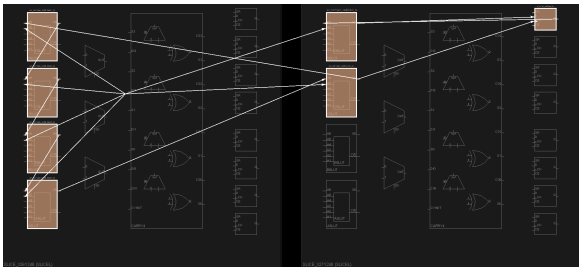

Fig. 3.4: Improved LUT-based RO placement example

RO Array Placement on FPGA

Figure 3.5 shows the manual placement of one RO PUF. To maintain the randomness and uniqueness, all the timing critical FFs and LUTs must be placed carefully. 128 ROs

are listed in six columns regularly in order to maximize the manufacturing differences between PUFs. Between columns, two slices are reserved for the relevant logic between ROs and the control unit. LUTs and FDs of the finite-state machine (FSM) are placed at the bottom of the RO array. Other primitives with no effects on the performance are placed by Vivado automatically.

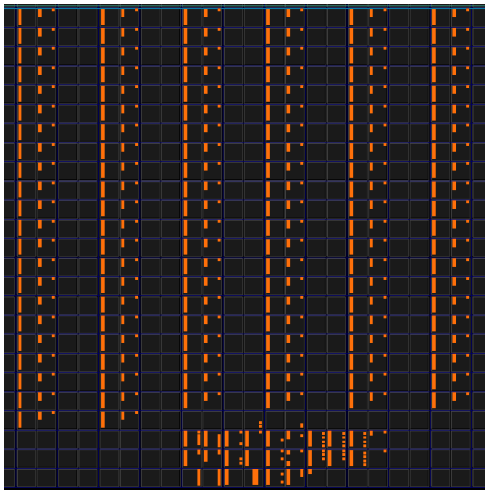

Fig. 3.5: RO PUF manual placement overview

Automatic Constraints Generation

However, manual placement of PUFs can be very time-consuming. In our example, each RO array contains 768 LUTs and 128 FFs. If the hardware specification is modified, it may lead to the reconstruction of the architecture. As a result, most of the cells have to be placed again. To make it progress more efficient, we developed an automation code generation process in Figure 3.6. The hardware specification module defines the length of CRPs, performance requirements, and usable platforms. The information

is translated to acceptable format of Shell script so that the code generation script can figure out the scale of RO array. With pre-defined templates, we are able to generate the code and constraints automatically.

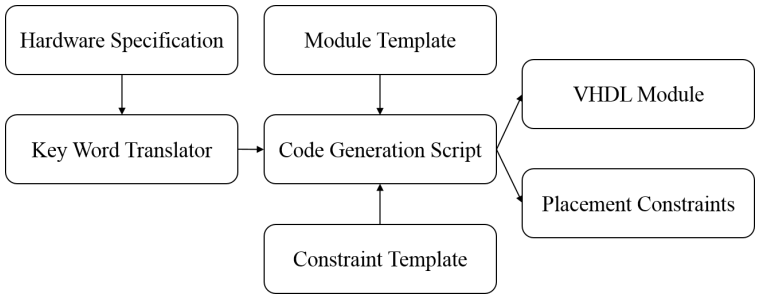

Fig. 3.6: RO PUF code generation

Manually Rounting Consideration

On the other hand, improper routing also affects the uniqueness of PUFs. Though our LUT and FF placement fixes the RO routing path, the other parts are routed by Vivado automatically. The critical paths are the RO array enable and the counter enable. If the RO array enable has a long delay while the counter enable has a short delay, it indicates that the timing to generate RO pulses is postponed and the counter is enabled in advance. In consequence, the window for measurement may not match the RO output well, which leads to potential issues of uniqueness and bias. To eliminate the effect of expected delay, we adjust unsatisfied wire delay by changing paths manually in Vivado implementation and saving it as constraints.

Stability Analysis

A key metric of any PUF is its reliability. In order to evaluate the stability, we have implemented three RO PUFs on the FPGA, which are based on Figure 3.2. We generated 100,000 random challenges and applied the challenge twice to each PUF. The two responses for each PUF were compared and the Hamming distance between the two responses were recorded. Ideally, there should be no difference between the two responses because the challenge was the same and the Hamming distance should be zero. Figure 3.7 shows the Hamming distance distribution for the three PUFs. Without any error correction or fault tolerance methodology, there are about 10 bit errors in each 128-bit response. It is clear that a 7.81% bit error rate is unsatisfactory for authentication. Without very strong ECC, we will have a high false negative rate. To address the instability of PUFs, we will introduce a phase calibration process in the next section.

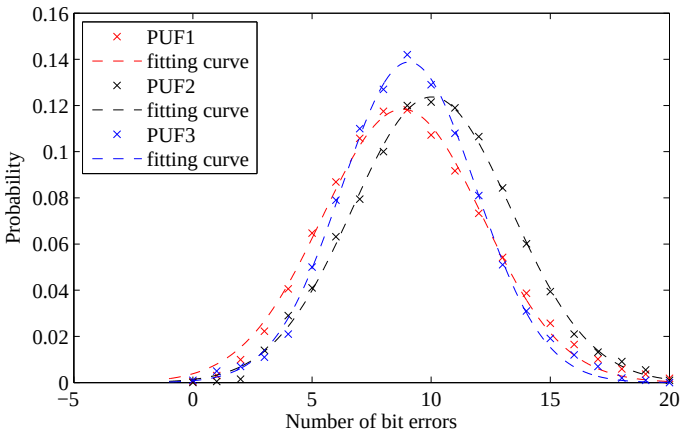

Fig. 3.7: Intra-PUF hamming distance without ECC

3.3 Phase Calibrated PUF

Improving the stability of PUFs requires an accurate measurement of the RO frequency. With the traditional RO PUF, theoretically, one could use a longer measurement time to count RO pulses and thus improve the measurement accuracy.. However, we found that extending the measurement time from 16 clock cycles to 512, in order to get larger samples, had limited improvement on the stability. Since the ROs are not driven by the system clock and each enable signal has its own delay, the output can be unstable when the counter is enabled and disabled [12]. As a result, the count may be off by one or two. Previously, we discussed adding a counter register to eliminate the unknown state, it does not address the correctness due to instabilities at the boundaries of the measurement cycle. Due to this unpredictable behavior, even though we tested some "stable" RO pairs 1,000 times and got the same results every time, the next measurement might still produce different value with a small probability. In this section, we propose an efficient solution instead of the repeated testing, which can solve this problem with limited cost.

3.3.1 Phase Calibration Process

Phase Calibration IP Core

The phase calibration process (PCP) is a critical part of our solution, which aims to measure the frequency of ROs fast and accurately. The basic idea is to shift the phase of the RO output signal in order to eliminate asynchronous timing measurement error.

To implement it on FPGAs, we use a primitive that offers a programmable delay function, i.e. Xilinx provides an input delay resource called IDELAYE2 [32]. It can be accessed directly from the FPGA logic and allows incoming signals to be delayed on an individual input pin basis. Figure 3.8 shows the IDELAYE2 primitive, which offers a variable delay mode that can control the delay value after configuration by manipulating the control signals CE and INC. When CE goes High, the increment/decrement operation begins on the next positive clock edge. The programmable delay taps in the IDELAYE2 primitive wrap-around. When the last tap delay is reached (tap 31) a subsequent increment function will return to tap 0. In Figure 3.8, a reset is detected (LD is High) in the first clock event, causing the output DATAOUT to select tap 0 as the output from the 31-tap chain. In the second clock event, a pulse on CE and INC is captured on the rising edge of C. This indicates an increment operation. The output changes without glitches from tap 0 to tap 1. In the third clock event, CE and INC are no longer asserted, thus completing the increment operation. The output remains at tap 1 indefinitely until there is further activity on the LD, CE, or INC pins.

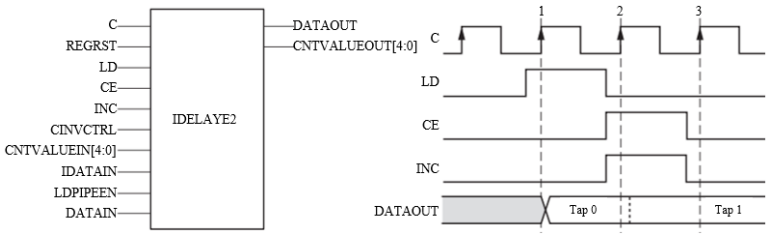

Fig. 3.8: Input delay resource and timing diagram

Architecture of Phase Calibrated PUF

The architecture of our phase calibrated PUF (PCPUF) is shown in Figure 3.9. It mainly consists of a 128-RO array, a control unit, a comparison unit, and pair of decoders, MUXes, IDELAYE2s, and counters. The tap control is to control the delay of IDELAYE2. Since PCP requires a strict RO enable timing, only one selected RO pair is enabled in the measurement period. Otherwise, the control unit resets all the ROs.

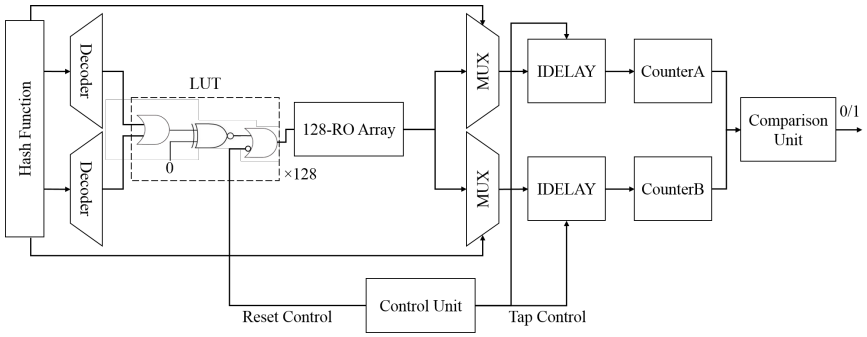

Fig. 3.9: Architecture of PCPUF

PCPUF Algorithm

Algorithm 4 shows the details of the PCP. In order to generate a 1-bit response, the first two 7-bit values of the challenge register are loaded to the register *addr_A* and register *addr_B*. After checking the values, we reset *CounterA* and *CounterB*. The tap value is loaded to IDELAYE2 primitives before being added by 1. By default, the tap value is 0. When the delay is set, the control unit enables the selected RO pair and starts testing. Each measurement takes 16 clock cycles. Then the tap value is added by 1 and the new measurement starts. Between two measurements, the selected RO pair is kept

disabled before the delay is set. When we complete the test from tap 0 to tap 31, a 1-bit output is generated by comparing the counter values. Then the challenge register shifts its value and launches the next response generation. Though the PCP slows down the response generation comparing to the traditional RO PUFs, the PCPUF still keeps an output speed of 625,000 bit/s.

3.3.2 Analysis of Phase Calibration Efficiency

Considering the calculable factors of FPGA, the accuracy of frequency estimation by using the phase calibration process depends on the skew rate, threshold voltage, and the relevant signal cycle. The skew rate and the threshold voltage determine the skew between the system clock and the counters. When the control unit enables or disables the counters at the beginning or in the end of the phase calibration process, the RO outputs may be changing between the voltage thresholds, $V_{OH\ min}$ and $V_{OL\ max}$. This may lead to a metastable state of the counter value. According to the I/O buffer specification (IBIS) model of the Kintex-7, if the FFs are in the same bank, the skew should be 50 ps to 100 ps. The internal signal voltage of the Kintex-7 is designed to swing between -0.5 and 1.1 volts, with anything below 0.4 volts considered a '0', and anything above 0.7 volts considered a '1' [33, 34]. In the worst case, the metastability duration of each rising edge or falling edge is 18.75 ps. Since the RO output phase at the beginning of the window is unpredictable, we can calculate the probability of the metastability occurrence in Figure 3.10 as

Algorithm 2 PUF phase calibration process

1: **procedure** PCP
2: $tap \leftarrow 0$
3: *loop 1*:
4: reset *CounterA*, *CounterB*
5: load *addr_A*, *addr_B*
6: **if** $addr\_A = addr\_B$ **then** $addr\_B \leftarrow addr\_A + 1$
7: *loop 2*:
8: load tap to $IDELAYE2$
9: $tap \leftarrow tap + 1$
10: enable $RO(addr\_A), RO(addr\_B)$
11: *loop 3*:
12: enable *CounterA*, *CounterB*
13: read $RO(addr\_A), RO(addr\_B)$
14: **if** *count cycle* < 16 **then goto** *loop 3*
15: keep *CounterA*, *CounterB*
16: disable $RO(addr\_A), RO(addr\_B)$
17: **if** $tap < 32$ **then goto** *loop 2*
18: **if** $CounterA > CounterB$ **then**
19: output 1
20: **else**
21: output 0
22: shift *Challenge*
23: **goto** *loop 1*.

$$p_{rising\_edge} = p_{falling\_edge} = \frac{18.75ps \times 2}{4210ps} = 0.00891 \tag{3.1}$$

in which 4210 ps is the average RO cycle. Though the rising edge and falling edge have the same probability, they are not independent events for any RO within one tap delay. Now we focus on the entire process in Figure 3.10. We create a window of 16 system clock cycles (80 ns), within which the counters of the RO PUF become active. Due to the wire delay, the selected RO pair may not be enabled at the same time. The delay between each RO and the related counter depends on a tap ranging from 0 to 31. Since each taps of IDELAYE2 offers 78 ps delay in the normal environment, we can have as much as 2.418 ns delay of the RO output, which is long enough to find the metastability. Ideally, these 32 taps phase shift will not affect the RO frequency measurement. However, some measurements show different counter values due to the metastability. According to our simulation, one metastability occurrence at one edge of the window has a probability of 0.00775. The probability that two metastability occurrences at both edges of the window is 0.00113. There is no possibility of any more unpredictable states as the window only has two edges. Thus the expected value of metastability is

$$E[X] = 1 \times 0.00775 + 2 \times 0.00113 = 0.01116 \tag{3.2}$$

Therefore, among the 32 taps shift cases, most of the counter values reflect the

frequency of the selected RO correctly. The metastability only happens with a very low possibility.

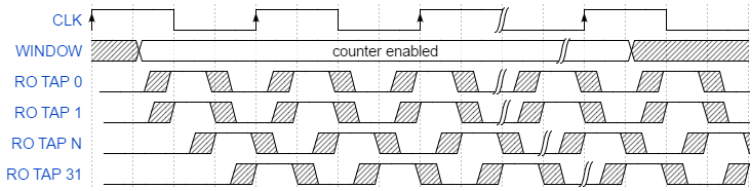

Fig. 3.10: PUF phase calibration timing

Apart from the calculable factors, measurement errors are also generated by the noise margin, which is the amount by which the signal exceeds the threshold for a proper '0' or '1'. Being affected by the uncontrollable factors such as temperature, voltage, and aging effects, we can hardly calculate the fault tolerate capability of phase calibration process directly [35]. But we are able to obtain the probability of bit flipping after PCP through large amounts of tests. Next we estimate the efficiency of PCP by computing the intra-PUF hamming distance.

3.3.3 Estimation of Intra-PUF hamming Distance

The bit error rate is one of the most critical characteristic of PUFs. To estimate the RBER of PCPUFs, we need to know the intra-PUF hamming distance first, which can be calculated through the distributions of RO frequencies and bit flipping rate (BFR). As we mentioned, when the frequencies of two ROs are closer, temperature, voltage, and aging effects during operation lead to higher BFR. We begin with measuring the rising edges of ROs in the unit time and show the probability distribution in Figure 3.11.

The unit time is set to 80 ns (16 clock cycles). Since we add the measured data from tap 0 to tap 31 and compute the average value, the results may not be integral. A fitting Gaussian curve is generated according to the discrete data, which is used to provide an ideal discrete distribution $D(f)$ for our computation.

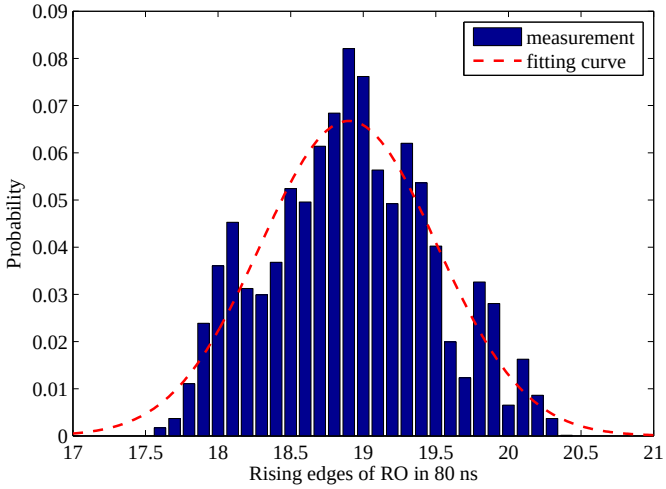

Fig. 3.11: Distribution of RO rising edges on average

We define a specific bit flipping rate λ according to the difference of the discrete data. Marked as f_1 and f_2, the frequencies of two ROs determine the probability of bit flipping. A measured $\lambda(|f_1 - f_2|)$ in Figure 3.12 shows the statistical BFR results, which follows part of a normal distribution $3.61 \times 10^8 N(-12.99, 1.88^2)$. When two ROs have a $|f_1 - f_2|$ larger than 1, their output is very stable. However, the BFR increases sharply when the difference range is less than 0.5. To further improve the reliability of PUFs, these RO pairs can be blocked from the RO pair selection sets.

For any single bit, the probability of error occurrence can be computed as

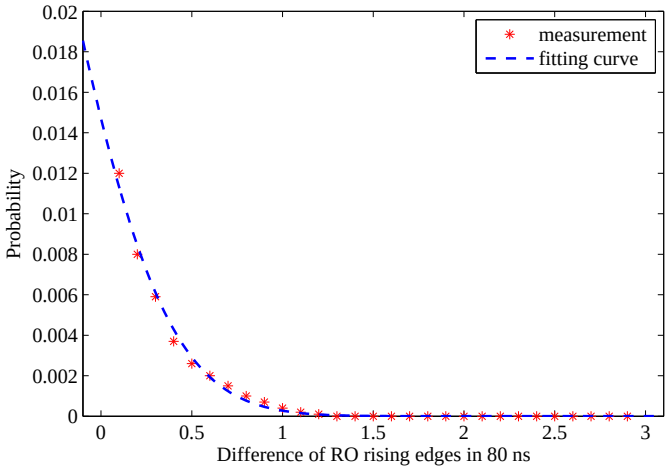

Fig. 3.12: Distribution of 1-bit RO output flipping rate

$$\sum_{f_2=0}^{\infty}\sum_{f_1=0}^{\infty} D(f_1)\times D(f_2)\times \lambda(|f_1-f_2|) = 0.0029 \tag{3.3}$$

The probability that 128-bit responses contain exactly n bit errors is

$$\binom{128}{n} 0.0029^n (1-0.0029)^{128-n} \tag{3.4}$$

According to Equation 6.5, the ideal intra-PUF hamming distance is shown in Figure 3.13. Therefore, we can calculate the average number of errors, which is 0.3688 in a 128-bit response.

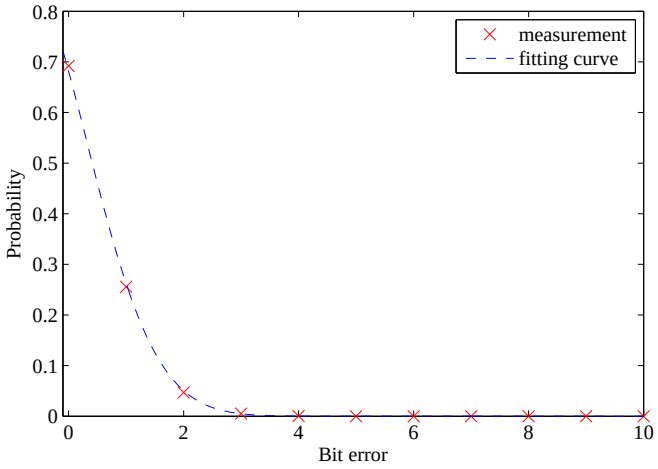

Fig. 3.13: Expected Intra-PUF hamming distance

3.4 PCPUF Evaluation

3.4.1 Experimental Setup

In this section, we present the measured data from six implemented PCPUFs on two KC705 boards, as shown in Figure 6.5. The temperature of FPGAs is controlled at 30°C according to the on-chip sensor and the voltage is 1.008V.

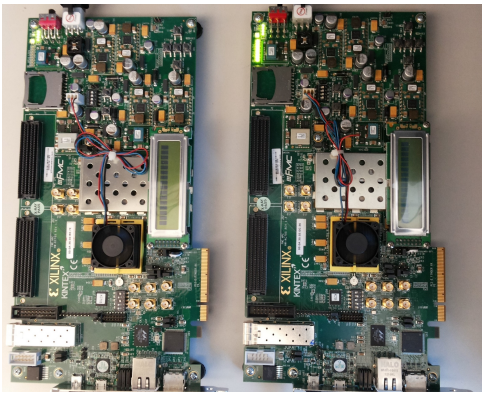

Fig. 3.14: PCPUF tests on KC705 boards

3.4.2 Hardware Resource Utilization

Table 6.3 lists the post-implementation primitive utilization of the traditional RO PUF and the optimized PCPUF. We exclude UART and FIFO module in the design to make a fair evaluation. The overall utilization of our PCPUF are 0.58% of the look-up tables and 0.05% of the flip-flops on the Kintex-7 FPGA. This is significantly low-cost compared to the traditional RO PUF design. If we apply the architecture in Figure 2.1, only 35 arrays can be placed on a Kintex-7 FPGA, which means four FPGAs are required to implement a 128-bit RO PUF. Since the shift register or hash function is used in the design, only one array is needed to generate multi-bit responses.

3.4.3 Randomness

We applied 15 NIST randomness tests to evaluate the randomness of our PCPUF responses. The p-values and proportions are listed in Table 3.2. As the p-values are all larger than 0.01, we accept the responses as random [36]. Though we use a pseudorandom number generator to generate challenges, the proportions show that at least 98% sequences pass the tests.

3.4.4 Uniqueness

The evaluation of the uniqueness of PCPUFs is shown in Figure 3.15, which compares three groups of 100,000 128-bit responses. The responses are generated by downloading the same bitstream to two different Kintex-7 FPGAs. The fitting curves have a μ

Table 3.1: FPGA Resource Usage of the Traditional RO PUF and the Optimized PCPUF Implementations

Ref Name	RO PUF	PCPUF	Functional Category
LUT1	514	2	LUT
LUT2	268	857	LUT
LUT3	175	55	LUT
LUT4	52	94	LUT
LUT5	273	30	LUT
LUT6	293	279	LUT
MUXF7	0	34	MuxFx
MUXF8	0	16	MuxFx
FDCE	20	16	Flop & Latch
FDRE	430	195	Flop & Latch
FDSE	1	1	Flop & Latch
CARRY4	7	9	CarryLogic
IBUF	4	4	IO
OBUF	2	2	IO
BUFG	1	1	Clock
IDELAYE2	0	2	IO
IDELAYECTRL	0	1	IO
Slice	647	365	CLB Slice
Utilization	1.26	0.71	%

Table 3.2: Results of NIST Randomness Tests

Statistical Test	P-value	Proportion
frequency	0.066882	0.98972
block frequency	0.213309	0.98972
cumulative sums	0.534146	0.98978
runs	0.739918	0.99075
longest run	0.031497	0.98939
rank	0.219646	0.98988
FFT	0.392456	0.98991
non-overlapping template	0.122325	0.98940
overlapping template	0.062947	1.00000
universal	0.599114	1.00000
approximate entropy	0.637119	1.00000
random excursions	0.778616	0.98902
random excursions variant	0.137809	0.98905
serial	0.039329	0.99161
linear complexity	0.534146	1.00000

converging to 64 - thus, there are 50% bits flipped on average. Therefore, the responses are unique. Moreover, we have compared the responses from six PUFs on two FPGAs and they all follow a normal distribution.

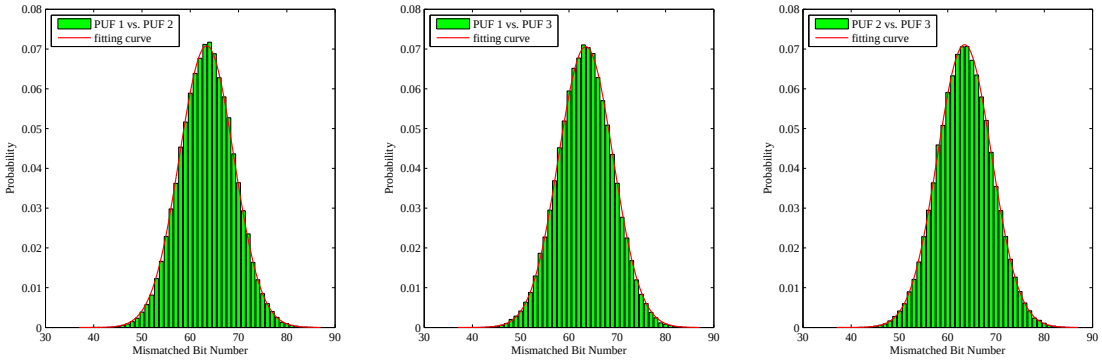

Fig. 3.15: Inter-PUF Hamming distance

3.4.5 Stability

As shown in Section 3.3, the average number of errors is 0.3688 in 128-bit responses, which means the theoretical bit error rate is 0.29%. To verify that, we generated challenges with a pseudorandom number generator. Each challenge is used twice in order to provide two responses for comparison. The test was repeated 100,000 times. Figure 3.16 shows the measured intra-PUF Hamming distance of our PCPUFs. Compared to the corresponding results with no ECC and phase calibration in Figure 3.7, the fitting curves for PCPUFs have obvious shift to the left and approach the ideal bound in Figure 3.13.

Table 3.3 shows the average bit error rate of three shift register based RO PUFs and six PCPUFs. Compared to RO PUFs, PCPUFs have obvious improvement on the response stability. Though the bit error rates are higher than the ideal bound, all the results are smaller than 1.00%. This is still acceptable if we take the measurement

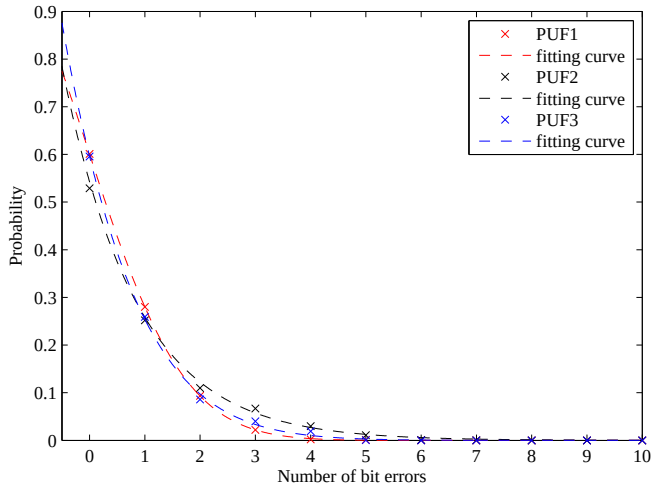

Fig. 3.16: Intra-PUF Hamming distance with phase calibration

errors into consideration. Since we do not apply ECC in our PUFs, the bit error rate is low enough for authentication purposes. Within the temperature range of normal testing environment, we observed no significant changes of the bit error rate.

Table 3.3: Bit Error Rate of RO PUFs and PCPUFs

FPGA 1	BER	FPGA 1	BER	FPGA 2	BER
RO PUF 1	7.10%	PCPUF 1	0.54%	PCPUF 1	0.37%
RO PUF 2	8.09%	PCPUF 2	0.85%	PCPUF 3	0.70%
RO PUF 3	7.13%	PCPUF 3	0.62%	PCPUF 2	0.91%

3.4.6 Bias

Ideally, there should be 50% 1's and 50% 0's in a response. However, our RO PUF implementation on Kintex-7 FPGAs have a bias ranging from 41% to 58% according

to our tests. As a comparison, we list the bias of PCPUFs in Table 3.4. Each result is based on the mean of 100,000 128-bit responses. Without any post-processing, all six PUFs produce bias within ±2% of ideal, which prove the effectiveness of the phase calibration on improvement the RO PUF bias.

Table 3.4: Bias of RO PUFs and PCPUFs

FPGA 1	Bias	FPGA 1	Bias	FPGA 2	Bias
RO PUF 1	40.85%	PCPUF 1	49.21%	PCPUF 1	48.72%
RO PUF 2	47.10%	PCPUF 2	49.77%	PCPUF 3	51.08%
RO PUF 3	58.42%	PCPUF 3	51.35%	PCPUF 2	50.14%

3.5 Conclusions

In this chapter, we demonstrated a practical design and implementation of RO PUFs on FPGAs. The phase calibration process focuses on improving the frequency measurement techniques in the crossing clock domain area. By involving phase calibration, the 7.81% bit error rate is reduced to less than 1%. We also evaluate other properties of PCPUF, including randomness, uniquess, and bias.

Chapter 4

Secure Authentication Solution with PUF Response Instability

4.1 Introduction

Authentication is widely applied in the digital world, which poses special problems with electronic communication and supply chain device identification. It has vulnerabilities to man-in-the-middle attacks and counterfeiting. Previously we introduce a stable PCPUF design. Based on that, we will discuss a specific authentication application using the instability of PCPUF in this chapter. Unlike the normal usage of PUF response, this work converts the unstable bits to the stable output. It is noted that the method can be applied to other types of PUF as well. Furthermore, an obfuscation system is provided to enhance the security of authentication solution, which also takes advantage of the unstable response in PUFs.

4.2 Unstable-Stable Response

Modeling and machine learning attacks are well-known for PUF adversary. The successful prediction is due to the accurate calculation and measurement of physical pa-

rameters such as delay and frequency. To achieve a secure PUF-based solution against those attacks, we propose a novel method to generate responses, which uses the instability of PUFs. Since the unstable bits in the RO PUF response are hard to be predicted, they provides better security feature than the stable ones. However, those bits have been only considered in the random number generation application yet. For other scenarios, only stable bits are usually required. In our work, we find the possibility to transfer the unstable bits to stable bits and apply them to PUF responses, which is present as unstable-stable response (USR), a more stable and secure response.

USR Generation

Figure 4.1 shows how to generate USR with a PCPUF. We set another hash function module out of the PCPUF, which can provide multiply sub-challenges with only one challenge input. Once a sub-challenge is created, it is sent to the challenge register for temporary storage. By reading the value from the register, PCPUF selects two RO pairs and keeps outputting the 1-bit response to the first address of the unstable bit counter for 1,000 times. The accumulated value in the counter reflects the stability of this output. A number that close to 0 or 1,000 is known to be stable while 500 is regarded as the most unstable response. With the help of its internal hash function module, the model generates a 128-bit raw response and stores it in the counter. Since PCPUFs provide good stability in their responses, the obvious unstable bits are so limited that less than one bit can be found in 128 bits. In the unstable bit counter, we set the unstable bits as

‘1’ and stable ones as ‘0’. The counter records the number of ‘1’s and sends the 128 bits to the response register. Before the bias reaches the ideal fifty-fifty, the counter enables the hash function module in order to launch the next progress with a different internal challenge. Another raw response is generated and transferred to a internal response in the counter. We apply the bitwise OR operation to this internal response and the one that stores in the response register. After updating the register value, the new response should have a higher probability to contain more ‘1’s. By repeating the same process, the response in the register reaches a balance between ‘1’ and ‘0’ finally. Now it is ready to be served as the final response.

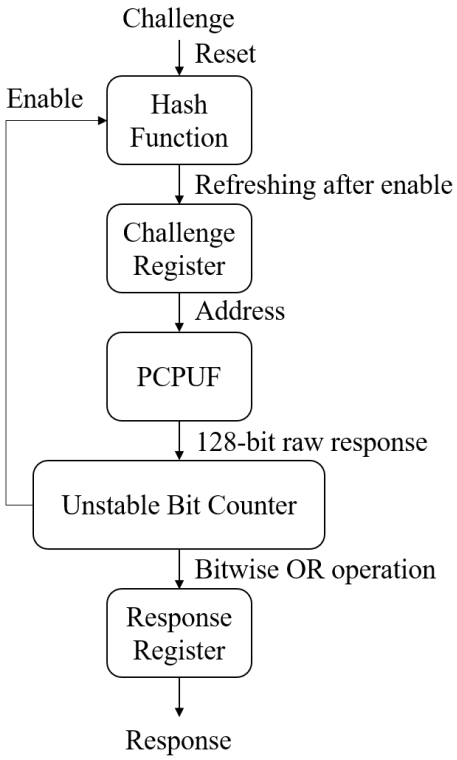

Fig. 4.1: Generating stable responses by using unstable raw responses

Proof of Theoretical Threshold

Note that the current design cannot guarantee a static threshold of how many times the process should be executed. The first reason is that the unstable bits selected by the hash function module are not controllable. Secondly, the '1's in two responses may be overlapped. Thus, the amount of '1' is reduced by the OR operation. To estimate the threshold, we define u as the number of unstable bits in 128-bit internal response. The probability that a certain bit is not set to '1' in one process is $1 - \frac{u}{128}$. Since there are t loops, the probability that the bit is not set to '1' by any loop is

$$p_0 = (1 - \frac{u}{128})^t \tag{4.1}$$

The probability that it is '1' is therefore

$$p_1 = 1 - (1 - \frac{u}{128})^t \tag{4.2}$$

To eliminate the bias, t should satisfy

$$\binom{128}{64} {p_1}^{64} {p_0}^{64} = 1 \tag{4.3}$$

By solving the equation, we are able to set a reasonable threshold in order to control the

bias. When the unstable bits is 0.5%, t is about 177. Though it is possible to control the bias with a dynamic adjustment, the required logic resource will increase significantly.

USR-based Authentication

As a basic PUF application, authentication does not require very low bit error rate of the PUF responses. Thus, we can apply our USRs directly to the solution. For the simplest authentication, we compare two responses and check if the mismatched bits are larger than a threshold t. In this application, false negative rate and false positive rate are critical for performance evaluation. Given a bit error rate e in 128-bit USRs, the false negative should be

$$f_n = \sum_{n=t+1}^{min\{m,128\}} \frac{\binom{16384-m}{128-n}\binom{m}{n}}{\binom{16384}{128}}, m \approx 128^2 * e \tag{4.4}$$

On the other side, the false positive is

$$f_p = \sum_{n=0}^{t} \binom{128}{n} (1-p)^n p^{128-n} \tag{4.5}$$

in which p is the probability of a random collision in 128 trials. Ideally, the value is close to 0.5.

Apart from the accuracy, we also need to estimate the cost of this solution. One important aspect is the utilization of memory space. In the simplest solution, we do not require ECC as the bit error rate in USRs is very low. Thus, there is no additional

overhead except the CRPs. Given the CRP length of n, the memory cost is $2n$ bits per CRP. This is acceptable for small amounts of authentication requests. However, as the requests grow, the memory cost increases linearly. A lightweight solution is to record the instability information of PUFs instead of CRPs. The assumption is that we trust the security of the database, which contains some sensitive information of PUFs. In that case, we can calculate the USRs with the known hash function, instability RO pairs, and the corresponding challenges.

As we mentioned above, an effective solution is based on the assumption that the database is secure. However, we must consider the scenario that the information is disclosed to attackers. The typical solution is to encrypt the database. Alternatively, we can choose a solution that combines the ECC and Bloom filter [37], which is both secure and efficient.

4.3 Obfuscation-based Authentication

Since some authentication applications require higher security level, we provide another scheme for our PUFs with a strong obfuscation. Our prime goal is that even if a current response is disclosed, it does not leak non-negligible information about the next responses with the same challenge, which means attackers cannot use replay attack to pass the authentication when a repeated challenge is applied. Secondly, the scheme has strong modeling attack resistance by involving unstable bits in the responses, but the authentication accuracy will not be affected in the database. Furthermore, though probe

attacks can obtain the information, the final response is still unpredictable.

Obfuscation Architecture and Algorithm

The scheme starts from the N-bit challenge register in Figure 4.2. Before launching the challenges, a PCPUF stability model is created in the hardware memory. It records the stable RO pairs and unstable ones. After a challenge is received from the database, the register value will not be updated until the final response is generated. In the first loop, the challenge is hashed by a one-way hash function module with an initial seed. The hash value is sent to the secure address generator as a secure challenge. By mapping the generated addresses to the PCPUF stability model, we select $N-1$ stable bits and one very unstable RO pair. We enable the unstable RO pair in order to generate 1-bit output for obfuscation. The position of the inserted unstable bit depends on the hash value. The N-bit temporary response is saved in *RegisterA*. Then we apply exclusive or operation to the value between the *RegisterA* and *RegisterB*, and store the result in *RegisterB*. In the first loop, the values in *RegisterB* is 0 as it is reset along with a new challenge. Since the secure address generator knows the selected $N-1$ stable bits, they are sent to the one-way hash function module and hashed as the secure challenge of the second loop. Before the hash operation, 1-bit '0' is added to the end of the stable bits to ensure that the input length is N-bit. This process repeats T times to generate the final response. In the loop t, the secure address generator still select $N-1$ stable bits and one unstable RO pair, which is stored in *RegisterA*. But there are t accumulated

unstable bits in *RegisterB*, given the different inserted positions by the secure address generator. Thus, the final response contains T unstable bits.

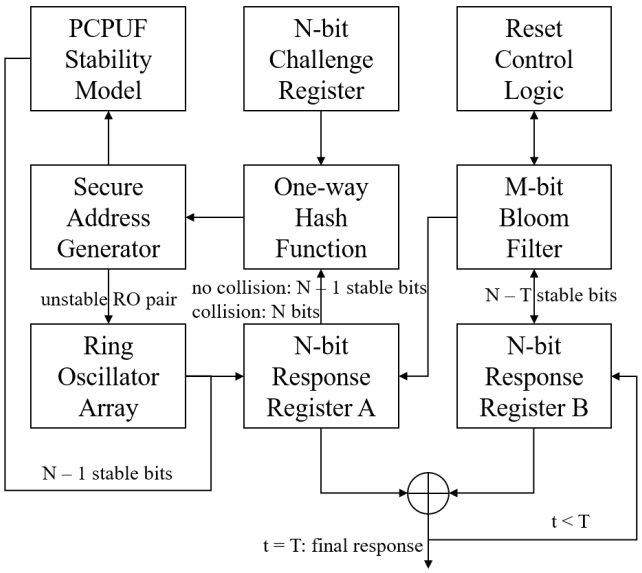

Fig. 4.2: Secure obfuscation-based authentication solution

To check the collision, we involve a Bloom filter and its reset control logic module in our scheme. A Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set. With an obfuscation bit mask from the secure address generator, the $N - T$ stable bits are sent from *RegisterB* to the Bloom filter. After being added by T-bit '0's, the string is hashed by k hash functions and the hashed values are stored in a M-bit non-volatile memory (NVM) Bloom filter. The NVM should be tamper resistant against any malicious reset. Whenever a final response is generated, we check the Bloom filter first to ensure that the response has never been released before. If there is no collision, we update the Bloom Filter with the k hash values and output *RegisterB* as the final response. However, Once a collision

is found, the final response will be stalled in *RegisterB*. As a further obfuscation, we rerun this process for T times to generate a new response. However, this time we switch the hash function input to the entire *RegisterA*, which means the unstable bit is included in the one-way hash input. After T loops, the final response will hardly disclose any information of the PUF state and pattern.

Due to memory space limitation, it is not possible to store infinite hash values in the NVM. Therefore, we use a reset control logic to cleanup the Bloom Filter when the false positive reaches a preset threshold. According to our scheme, the PUF is not allowed to generate the same response twice within the threshold times. Modeling attacks are also inefficient as stable response bits and internal state is obfuscated by the unstable bits, one-way hash function, and the Bloom filter. Even if the Bloom filter is reset, the pattern cannot be detected due to the T times one-way hashing. The complexity of our scheme also makes probe attacks useless since the next final response is unpredictable given a known PCPUF model, Bloom filter content, and the current register value. Since the attackers cannot try all 2^N input patterns when N is large, our scheme is secure.

Database Consideration

To authenticate our PUFs in a secure way, we avoid recording any data in the database that may disclose the information of the PUFs. Instead, we use Bloom filters for authentication, in which the hash functions provide strong resistance against software reverse

Algorithm 3 Secure authentication scheme

procedure OBFUSCATION

loop 1:

reset *RegisterB*, t

ChallengeRegister ← *new challenge*

loop 2:

if $t = 1$ **then** hash(*new challenge*(N))

else if $t = T + 1$ & no collision **then goto** *loop 3*

else if $t = T + 1$ & collision **then** output *RegisterB* **goto** *wait*

else if *no collision* **then** hash(*RegisterA*($N - 1$)+'0')

else if *collision* **then** hash(*RegisterA*(N))

generate $N - 1$ stable bits with the PCPUF model

generate 1 unstable RO pair with the PCPUF model

generate 1 obfuscation bit with the RO array

generate inserted position with the hash value

RegisterA ← N bits

RegisterB ← *RegisterA* $\oplus$ *RegisterB*

$t \leftarrow t + 1$

goto *loop 2*

loop 3:

receive *obfuscation bit mask*

BloomFilter ← *RegisterB*($N - T$) + T'0'

if *collision* **then** $t \leftarrow 1$ **goto** *loop 2*

else if *no collision* **then** update *BloomFilter* output *RegisterB* **goto** *loop 1*

engineering. When a PUF is manufactured, we test the PUF and store the PCPUF stability model to the database. Then we redo the response generation process by software, with our challenge set and the known initial seed in the PUF. Since we cannot predict the unstable bits in the response, all the possibilities must be generated by an authentication tree, as shown in Figure 4.3. The responses of $R1$ are generated after the first loop of calculations according to the challenge $C1$. $R1(0)$ and $R1(1)$ are based on the assumption that the obfuscating unstable bit is '0' and '1'. In the second loop, the responses of $R2$ are generated with the $N-1$ stable bits from $R1$ and the unstable bits lead to four combinations. By following this progress, we obtain 2^T possibilities of the final response. With the same challenge $C1$, the final response of this PUF must belong to the set of RT. Thus, all the responses in this set need to be recorded in the Bloom filter with $C1$ as one-time CRP. As we apply NONCE to the challenge set, it is not necessary to calculate and store the final response set of the second T loop. By mapping the response set with one-way hash functions, we can guarantee the data security in the database.

This security scheme sacrifices hardware resources however. For instance, after generating 1 million responses, the Bloom filter will be reset to maintain a false positive of 10^{-6}. The size of NVM and the number of hash functions are calculated as:

$$M = -\frac{n \ln p}{(\ln 2)^2} \approx 3.42\ MB \tag{4.6}$$

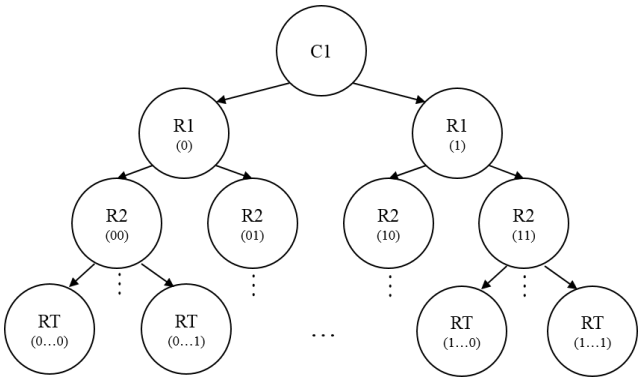

Fig. 4.3: PUF authentication tree in the database

$$k = \frac{M}{n} \ln 2 \approx 20 \tag{4.7}$$

In the database, the false positive rate of the Bloom filter affects the authentication accuracy. Thus, it needs to be much lower than that in the PUF. If we set it to 10^{-20}, the size of Bloom filter for 1 million CRPs is:

$$M' = -\frac{n \ln p}{(\ln 2)^2} \approx 11.43\ MB \tag{4.8}$$

Obviously, the cost is higher than the USR-based authentication solution. In summary, there is always a trade-off between the cost and the security. Those low-cost solutions are suitable for a large amount of chips while this scheme enhances the security level for some critical applications.

4.4 Performance Evaluation

4.4.1 USR Authentication Result

The performance of USR-based authentication is shown in Table 4.1. Compared to the measured raw response of PCPUFs, USRs have lower bit error rate. When we set the threshold to 20, all three PCPUFs have a low false positive and false negative rate in theory. We also verify the bound by measuring 300,000 CRPs on an FPGA. The results show that no false positive or false negative is detected.

Table 4.1: False Positive and False Negative of USR-based Authentication

FPGA 1	BER	t	False positive	False negative
PCPUF 1	0.30%	20	4.30×10^{-16}	$5.38 \times 10^{-22}\%$
PCPUF 2	0.38%	20	4.30×10^{-16}	$1.41 \times 10^{-19}\%$
PCPUF 3	0.31%	20	4.30×10^{-16}	$8.82 \times 10^{-22}\%$

4.4.2 Comparison with ECC Solutions

Table 4.2 shows the other authentication solutions with Hamming ECC and BCH ECC. Hamming(255,247) can correct one bit error in 128-bit responses while BCH(255,128,15) has a correct capability of as many as 15. However, the overhead of BCH ECC is much larger than that of Hamming ECC. We apply both ECCs to the shift register based RO PUFs and our PCPUFs. Compared to the poor performance on RO PUFs, a lower bit error rate of PCPUFs takes more advantages from ECCs. When those 100,000 CRPs

are tested with BCH ECC, no false negative is detected. Though it is hard to estimate the accurate false negative of BCH ECC based authentication on PCPUFs, we can compare the overhead of these solutions in order to find a better trade-off. The codeword of BCH ECC is almost the same size as the key, while our USR-based solution has no additional overhead at all. Therefore, USR-based authentication is recommended when the memory space becomes a critical bottleneck.

Table 4.2: Performance of ECC-based Authentications

PUF type	ECC type	overhead	False negative
RO PUF 1	Hamming(255,247)	8	1
RO PUF 2	Hamming(255,247)	8	1
RO PUF 3	Hamming(255,247)	8	1
RO PUF 1	BCH(255,128,15)	116	3.82×10^{-2}
RO PUF 2	BCH(255,128,15)	116	4.84×10^{-2}
RO PUF 3	BCH(255,128,15)	116	2.21×10^{-2}
PCPUF 1	Hamming(255,247)	8	1.21×10^{-2}
PCPUF 2	Hamming(255,247)	8	2.10×10^{-2}
PCPUF 3	Hamming(255,247)	8	1.59×10^{-2}
PCPUF 1	BCH(255,128,15)	116	$< 10^{-6}$
PCPUF 2	BCH(255,128,15)	116	$< 10^{-6}$
PCPUF 3	BCH(255,128,15)	116	$< 10^{-6}$

4.5 Conclusions

In this chapter, we demonstrated the potential of using instability of PUF responses as stable output. The unstable-stable response method transfers unstable bits to stable output. The results show that USR has better performance than the traditional ECC solutions. Another contribution is the obfuscation design of PUF responses. By applying the unstable RO pairs as stable responses and obfuscation, we improved the stability and security of PCPUF-based authentication.

Chapter 5

Ring Weight Algorithm for Lightweight Authentication

5.1 Introduction

As the integrated circuit industry grows, counterfeiting ICs have also been growing as an illegal means of profit taking. In response to this trend, authenticating untrusted ICs has become a critical and pressing issue. Many methods are already available to identify and authenticate ICs [38]. Among them, solutions based on PUFs show the high level of security due to their unpredictable response [1].

The high bit error rate in the responses of PUFs requires strong error correcting codes (ECC) to guarantee a low false negative rate in authentication processes [39]. However, a strong ECC requires large memory space and high calculation complexity. Fuzzy extractor is another solution with higher security level apart from ECC, which involves even larger overhead [8, 40]. As a result, it causes a trade-off between the failure rate and the overhead. Though it is necessary to apply traditional ECC or fuzzy extractor when using PUFs as key generation purpose, IC authentication process does not need to correct the errors in responses, but tolerate them.

In this chapter, we propose a novel algorithm that not only keeps false positive and false negative small, but also reduces the overhead and computation time. This algorithm calculates the density of 128 bits distribution in a response and selects certain information for authentication. By tolerating the weight and location shift of these values, errors in responses affect the accurate rate negligibly. Additionally, the encrypted helper data from each chip improves the performance in a practical and secure way. Thus, a larger set can effectively prevent cloned chips from colliding and being authenticated by mistake.

5.2 System Architecture

Figure 5.1 shows an on-board self-authentication architecture where IC PUFs can be authenticated on the board itself. The system is composed of an authentication device that can issue challenges and receive responses from an IC PUF and a database where the challenge and response pairs are stored. In addition, the data generated by the reliable responses can be sent to the memory for offline usage. In the figure, the authentication device is shown as a FPGA device, but could be implemented with any trusted device with compute capabilities as well. For self-authentication, the database should be stored on the board in some type of nonvolatile memory. If authentication is done offline, the database is stored in a secure server. The database is a critical part of the authentication process and it suffers from three problems – namely size, error tolerance, and security. The size of this database can be a critical issue as it needs to

store all challenge response pairs and the on-board storage may be limited. Since most current PUFs are not reliable – i.e., there is the possibility of bit errors, and we assume that the IC PUF has no built-in error correction, the authentication device and database must handle any errors in the response. Finally, security is a prime concern, because if the database is compromised, attackers can easily obtain the CRPs from the database and clone the chip with faked responses of PUFs.

We show three different approaches to address these issues – standard error correction with encryption, error correction with Bloom filter, and fault tolerant extremum matching. Depending on the approach, the data stored in the database will differ.

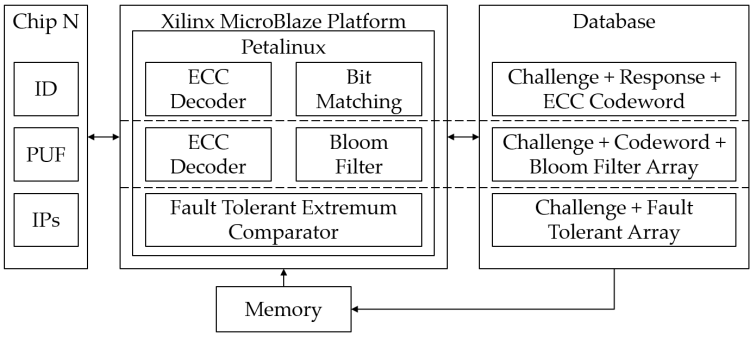

Fig. 5.1: The architecture of three IC authenticating solutions

5.2.1 Error Correcting and Bit Matching

The simple solution is to store CRPs and ECC code words in the database for authentication. Since the responses from the PUFs may contain a number of bit errors [41, 42], a strong ECC is required to guarantee a low false negative rate. However, this approach requires a larger database in order to hold the ECC code words. Moreover, it requires

encryption of the database which entails more computation effort as well as the difficulty of managing the encryption key.

5.2.2 Error Correcting/Bloom Filter Matching

In order to deal with the size of the database, one could use one of a number of lossless compression techniques. However, the database must be uncompressed eventually, meaning, while nonvolatile memory is reduced, the volatile memory must still be large enough to store the CRPs and ECC code words. Instead, for our application, we need a compressed data structure that can be queried without uncompression. A Bloom Filter is such a data structure: a space-efficient probabilistic data structure that can be used to test if an element is a member of a set [43]. The answer to this query can be either "possibly in set" or "definitely not in set", which means it can have false positives but never false negatives. A Bloom filter is a bit array with k different hash functions. Feeding an element to these k hash functions produces k array positions which can be set to 1. Querying for an element requires feeding it to the k hash functions and verifying that the resulting bit positions are set to 1. Thus, it provides faster detection of information without resorting to associative lookup buffers [44, 45]. To enhance the information security, one-way hash functions can be used instead [46]. Due to the one-way hash functions, it is computationally hard for attackers to retrieve original responses from the Bloom filter array and code words. In the context of IC/PUF authentication, a Bloom filter can be used to store the challenge-response pairs. As a result, the database is

composed of challenges, ECC code words, and the Bloom filter. While the Bloom filter reduces the size of the database significantly by reducing the size of the response space, the need to store ECC code words still requires a large dataset.

Consider a Bloom filter with a size of m bits and k hash functions. If n is the number of elements to be stored in the filter, the false positive rate, p_f, can be calculated as approximately [47]

$$p_f = (1 - e^{\frac{-kn}{m}})^k \tag{5.1}$$

For a given n and a desired p_f, the optimal m and k can be calculated by the following equations:

$$m = -\frac{n \ln p_f}{(\ln 2)^2} \tag{5.2}$$

$$k = \frac{m}{n} \ln 2 \tag{5.3}$$

As an example with $n = 10^6$, a 28755176-bit array and 20 hash functions are required to keep the false positive rate below 10^{-6}.

This solution calls for a strong BCH ECC due to the high bit error rate in responses [48]. On the other hand, it sacrifices time and memory space efficiency. For instance, to correct 12 errors in a 128-bit response, eight dimensions of Galois Field

and 92-bit code word are required, which means large overhead and long-time encoding and decoding. Further, if errors are more than the limitation of error correcting capacity, false negatives will occur due to the failure of error correcting and the false mappings into the Bloom filter.

5.2.3 Fault Tolerant Extremum Matching

The ideal solution is to use a strong input fault-tolerant matching algorithm to replace ECC. The fault-tolerant array stores limited information such that even if the array is disclosed, the responses are not available. The algorithm should be efficient while the security level remains high. The overhead should be small while the false positive and false negative are low. This algorithm is the main contribution of this chapter and is described in further details in the next section.

5.3 Novel Fault Tolerant Methodology

To avoid the large overhead of using ECC, we propose a novel fault tolerant authenticating scheme, which does not rely on ECC. It can achieve a better trade-off between false positive rates, false negative rates, the computing time and the memory and area overhead.

5.3.1 Ring Weight Algorithm

The ring weight algorithm (RWA) [37] implements a similar function as the Bloom filter plus ECC, but it requires very limited overhead for large input-fault-tolerance capabilities. We assume a 128-bit response is separated into 32 groups, which form an outer ring as shown in Figure 5.2. Each group contains a 4-bit value ranging from 0 to 15 and is marked with a location value by sequential order. According to our simulation results in Table 5.1, we select 4-bit groups as a trade-off among false positive, false negative, and overhead bits (maximum weight plus maximum location). The 32 location values compose the middle ring. The inner ring contains 32 weights ranging from 10 to 0. Each weight is used only by the corresponding 4-bit group. If these weights are not well-assigned, it will affect the authentication accuracy. Thus, we test linear and non-linear weight distributions and select the optimal assignment according to our simulation results. The overall weight of a location is calculated by multiplying each 4-bit value by the current weight value aligned with it and taking the sum. Clearly, the range of the overall weight is from 0 to 1800, since a response can be set from all 0's to all 1's. Figure 5.2 shows the ring in the 1st location and has an overall weight of 849. This overall weight is computed by $10 \times 10(1010) + 9 \times 5(0101) + ... + 0 \times 11(1011) + ... + 9 \times 7(0111)$ and is recorded in the first location of the middle ring. The same process is done for every position as we rotate the weight dial clockwise for the next number. Thus, in the second rotated position, the second 4-bit value (0101) now is aligned with the weight of 10. We follow the previous rules to compute the overall

weight: 10×5(0101)+9×5(0101)+...+0×15(1111)+...+9×10(1010). The new overall weight value is recorded in the second location of the middle ring. After rotating the weight dial 360 degrees, we can obtain 32 overall weights, among which the maximum and minimum value are recorded as the peak and valley value of the response. The corresponding peak and valley weight ring locations are also stored in the fault tolerant array as a criterion for judgment.

Table 5.1: Performance of Different Bit Group Selection

scheme	false positive	false negative	overhead bits
1-bit	31.95%	11.68%	15 + 7
2-bit	33.68%	10.84%	13 + 6
4-bit	35.72%	9.80%	11 + 5
8-bit	51.68%	5.05%	9 + 4
16-bit	74.56%	1.97%	7 + 3

A simple authentication method is to calculate the peak or valley information of a response and compare it with that in the fault tolerant array. As the errors increase, the overall weight and location value may not be fitted perfectly to the offline array value. Thus, the shifting of the location and weight value should be tolerated to some certain extents. By way of illustration, the offline recorded peak weight is 1138 and the related location is 17. However, the results calculated from the response turn out to be 1105 and 16 due to several bit errors. In this case, the response is rejected, and the device cannot

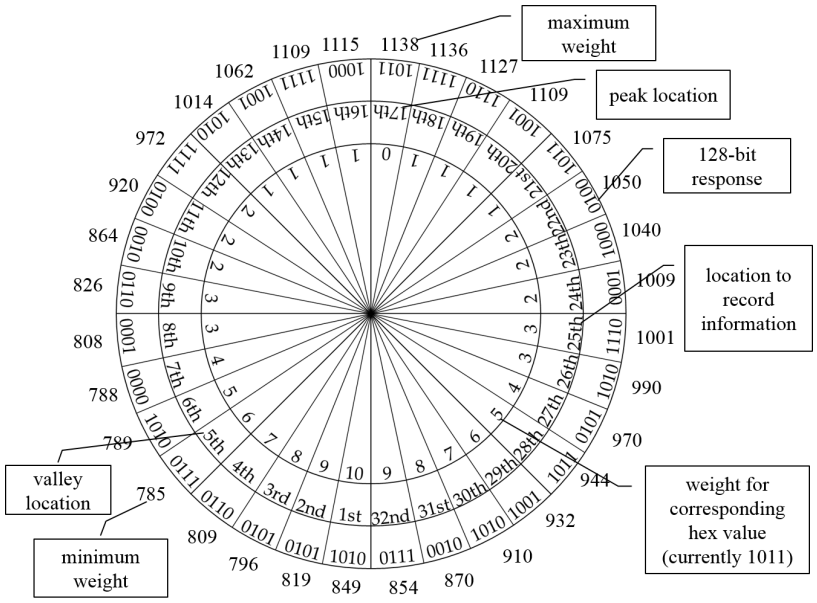

Fig. 5.2: Main structure of ring weight algorithm

pass the authentication process. In order to allow for errors, we can set a tolerance threshold where we accept deviations in the weight and location value. For example, if ±50 or ±1 are set as the acceptable tolerant range for weight or location value, the response can still be authenticated correctly by weight-tolerance or location-tolerance. However, allowing for tolerance thresholds introduces the possibility of false positives - i.e. responses that are authenticated even though they were not valid. For example, 1105 and 16 could be the weight and location calculated from an invalid response rather than due to errors in a valid response. Reducing the tolerance threshold ranges can reduce the incidence of false positives, but that would increase the probability of false negatives - i.e. responses that are rejected even though they were authentic. Thus, the tension is coming up with optimal values of ranges to minimize both false positives and

false negatives. In the following section, we discuss some of the theoretical foundations behind calculating false positive and false negative rates.

5.3.2 False Positive and False Negative

In this section, we provide the relationship between the parameters of RWA and the authentication results. In RWA, there are two parameters for fault tolerance usage: weight and location. The maximum and the minimum values can be used for authentication. In the aspect of the performance, false positive and false negative are the main factors of authentication process. Thus, three groups of choices generate eight combinations of simple authentication method.

Weight-based False Positive

We start the proof with the relationship between the maximum weight and the probability of false positive. Figure 5.2 shows a weight pattern ranging from 0 to 1800. $\{W_1, \ldots, W_n\}$ is a set that records the maximum weight values of non-error responses. For random distributed responses, the probability density function (PDF) of weight belongs to a certain family of distribution with a vector W'. The measurement error is given as $W - W'$. Thus, the likelihood function of the joint probability is

$$L(W') = L(W'; W_1, \ldots, W_n) = f(W_1 - W') \cdots f(W_n - W') \tag{5.4}$$

Maximum-likelihood estimation is applied as

$$\frac{d \ln L(W')}{dW'} = \sum_{i=1}^{n} \frac{f'(W_i - W')}{f(W_i - W')} = \sum_{i=1}^{n} g(W_i - W') = 0 \tag{5.5}$$

For each 4-bit group $\{r_n, r_{n-1}, r_{n-2}, r_{n-3}\}$ in a response, the weight value is calculated by

$$W = \sum_{i=1}^{32} (w_i \times (2^3 r_n + 2^2 r_{n-1} + 2^1 r_{n-2} + 2^0 r_{n-3})), n = 4i \tag{5.6}$$

If $\overline{W}$ is the arithmetic mean value of W, for the equation (5.6), with $n = 2$, then

$$g(W_1 - \overline{W}) + g(W_2 - \overline{W}) = 0 \tag{5.7}$$

Since the minuend is the arithmetic mean of W_1 and W_2, the function can be simplified as $g(-W) = -g(W)$. Further, let $W_1 = \ldots = W_m = -W$, $W_{m+1} = mW$, in which $n = m + 1$. An arithmetic mean value of 0 leads to

$$\sum_{i=1}^{n} g(W_i - \overline{W}) = mg(-W) + g(mW) = 0 \tag{5.8}$$

Thus, we have $g(mW) = mg(W)$. The only continuous function that satisfies this equation is $g(W) = cW$, which means the PDF of W is

$$f(W) = ae^{c(W)^2} \tag{5.9}$$

Normalized $f(W)$ follows the general normal distribution. The discrete function is described approximately as

$$p_\phi \approx \frac{1}{\sigma}\phi(\frac{W}{\sigma}) \tag{5.10}$$

Therefore, the maximum weight follows a normal distribution. To estimate σ and μ, we ran one million CRPs as a sample set. The bits in these responses are distributed uniformly to guarantee the accuracy. The simulated data with a fitted curve are shown in Figure 5.3. Since the normal distribution curve achieves 0.999 adjusted R-square with the simulation result, we can safely say that it represents how the maximum weight values distribute from 0 to 1800. The parameters of the normal distributions are represented as $N(1008, 106^2)$. In another words, the probability of maximum weight can be applied to that of any random 128-bit response.

To calculate the maximum weight-based false positive, we apply convolution to the curve in Figure 5.3. The convolution represents the collision rate between an invalid response and the actual valid response. The process is shown in Figure 5.4. The blue curve, representing the invalid response, starts shifting from the left side when μ_{blue} −

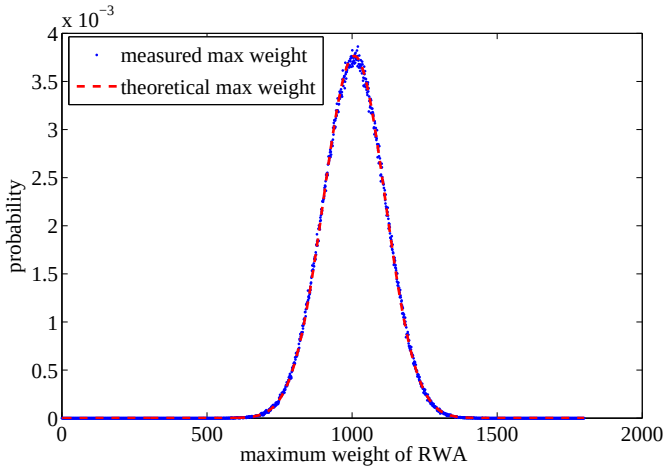

Fig. 5.3: The maximum weight distribution of RWA

$\mu_{red} = -1800$. At this point, the shaded area is equal to the probability of a collision with a value of $\Delta W_{max} = -1800$. As the blue curve shifts closer to the red curve, the shaded area increases as a smaller ΔW_{max} has a higher probability of collisions. After reaching the peak point $\Delta W_{max} = 0$, the probability decreases in a symmetrical way. The results of the convolution are shown in Figure 5.5. The figure also shows the result of a simulation of one million CRPs and one million random responses. As can be seen, the simulated values of ΔW_{max} are matched with the convolution data. Clearly, if we tolerate a wider range of ΔW_{max}, the integral and thus, the false positive probability is higher. For example, if the tolerant range is 0, the false positive rate is 0.027. It increases to 0.769 with an interval between -180 to 180 in Figure 5.6. A tolerant range between -500 to 500 covers most collisions, but also means the probability of a false positive is nearly 1. Thus, the tolerance setting need to be controlled in a reasonable range in order to remain effective at keeping both the false positive and false negative

rates small.

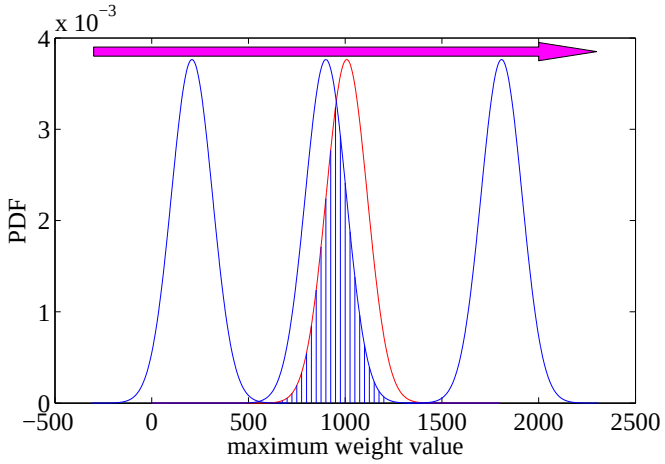

Fig. 5.4: Convolution of the maximum weight distribution

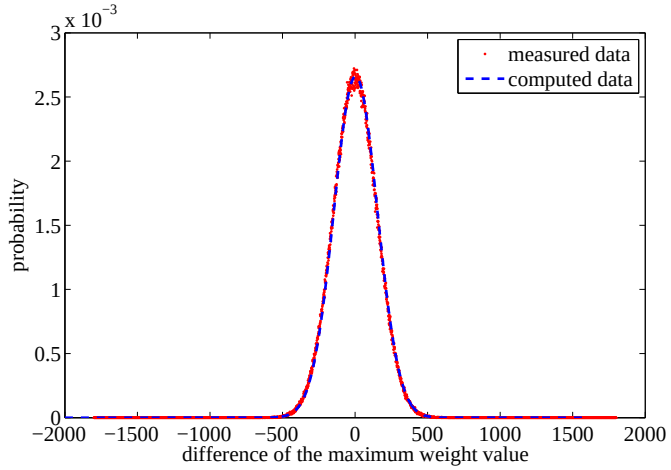

Fig. 5.5: Collision of the maximum weight difference value

The above proof can also be applied to minimum weight-based false positive. The curves in Figure 5.3 and Figure 5.7 show axial symmetry properties with the axis of $W = 900$. In Figure 5.8, the convolution results match the simulated minimum weight difference. Therefore, minimum weight-based tolerance has the equivalent collision

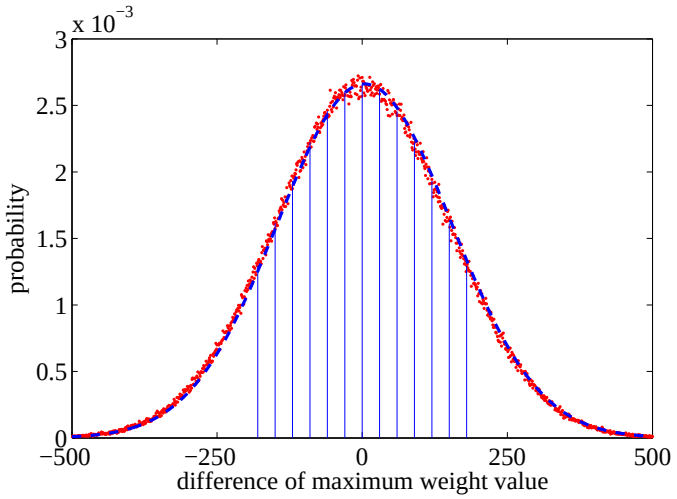

Fig. 5.6: Probability of the maximum weight based false positive rate as that of the maximum weight-based tolerance.

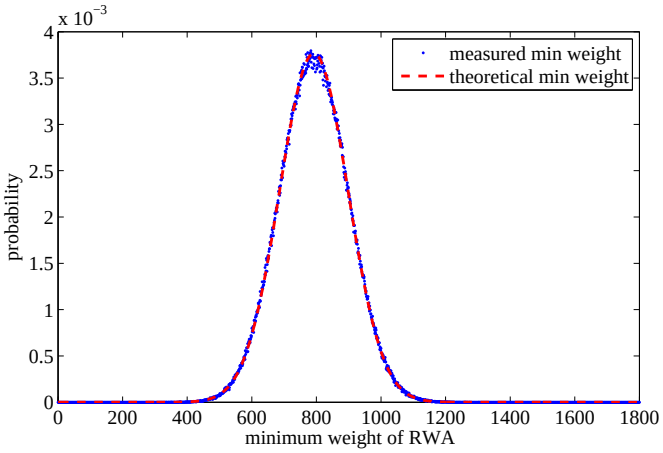

Fig. 5.7: The minimum weight distribution of RWA

Location-based False Positive

As we mentioned in the RWA section, the peak location L_{max} refers to the maximum location while the valley location L_{min} refers to the minimum location. Since they can

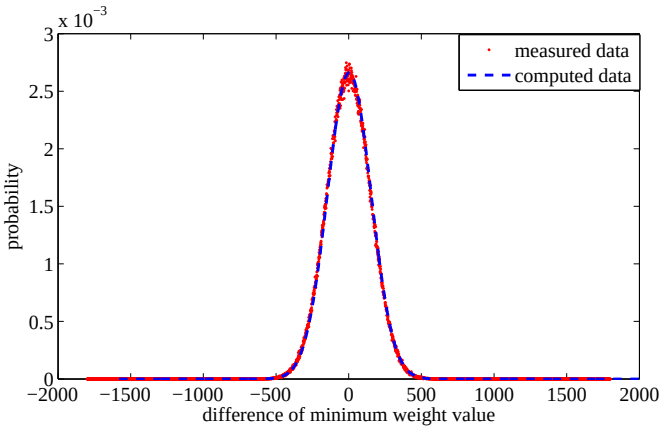

Fig. 5.8: Collision of the minimum weight difference value

be proved to have similarly effects on location collisions, only the peak location-based false positive is discussed in this section. This authentication is based on tolerating ΔL_{max} caused by the bit errors in responses, but it involves false positive as well. The collisions of ΔL_{max} are caused by uncertain locations of random responses. In Figure 5.9, we present the relationship between ΔL_{max} and the collision rate. Since we define L_{max} as $\{1, 2, \ldots, 32\}$, ΔL_{max} has an interval of $[-31, 31]$. With a fixed original location, the tested response has an equal chance to appear at any place in the ring. Thus, all the accessible locations are marked with the same probability unit $'1'$ while all the unavailable locations are $'0'$. The collision rate can be calculated according to the amount of $'1'$ in the shape of diamond in the diagram. The position of $\Delta L_{max} = 0$ obtains the highest p: 32 out of 1024. On its both sides, there are progressive decreases in probabilities. $\Delta L_{max} = -31$ and $\Delta L_{max} = 31$ only appear when L_{max} is 1 and 32. A comparison of the location collision is shown in Figure 5.10. We simulated one million

CRPs and generated a collision rate set, which overlapped the theoretical probabilities according to Figure 5.9.

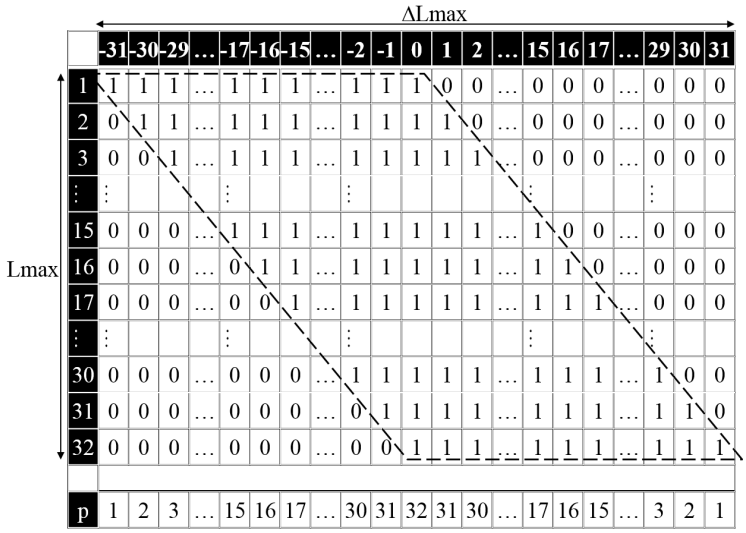

Fig. 5.9: Distribution of the maximum location difference

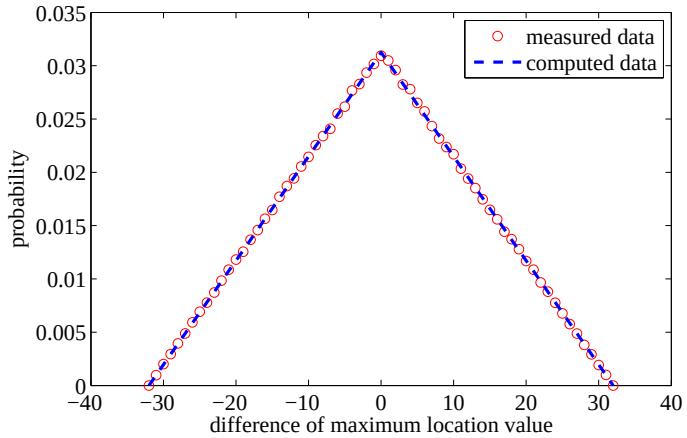

Fig. 5.10: Collision of the maximum location difference value

Notice that in the ring, ΔL_{max} follows a rule: $-31 = 1, -30 = 2, \ldots, -16 = 16, \ldots, -2 = 30, -1 = 31$. Thus, in Figure 5.10, the triangle from -31 to -16

can be flipped horizontally to the space from 1 to 16. The triangle from 17 to 31 can be flipped horizontally to the space from -15 to -1. As shown in Figure 5.11, by transferring ΔL_{max} into absolute value, the collision rate of location-based RWA becomes flat from -15 to 16. When tolerating one more location with the current location range, the collision rate increases by $1/32$. To reduce the false positive, we will discuss an optimization in the location shift scheme section.

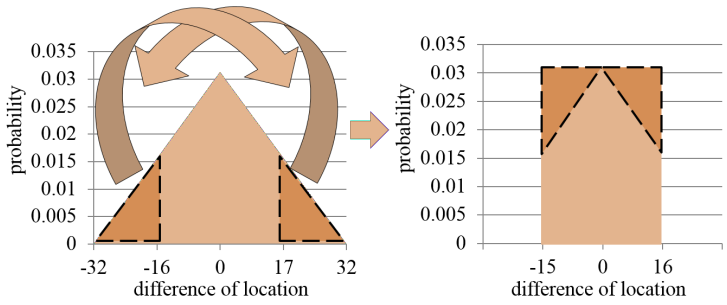

Fig. 5.11: Absolute value of the maximum location difference

Weight-based False Negative

We consider an unauthenticated 128-bit response with 12 bit errors, there are $2^{128} \times \binom{128}{12}$ bit distribution patterns. In other words, we need to run the program for 8.07×10^{54} times to calculate the true probability of weight-based false negatives. Instead, we estimate the probability by simulating one thousand samples to evaluate the distribution of ΔW_{max}. Each sample is a set of one million simulation results. One set refers to one certain response with average 12 bit errors, but the error is distributed in different patterns. Here, the number of errors is assumed to follow a normal distribution. The

examples are given in Figure 5.12. If a response is all $'0'$, it will only have negative ΔW_{max}. A response with all $'1'$ shows opposite features. However, most of the random responses fit a standard normal distribution with $\mu = 0$. All the sets together form a normal distribution in Figure 5.13. We can safely claim that, though it is difficult to calculate all the probabilities of weight-based false negatives, the results from the samples have statistical significance and can be regarded as a close approximation of the true probabilities. The simulated data matches the statistical curve in Figure 5.13. Most of the error cases can be tolerated with a ΔW_{max} between -275 to 275. The range is determined mainly by the number of errors, as shown in Figure 5.14. With more errors, the standard deviation, σ, becomes larger and the probability is less concentrated around 0. Thus, a wider ΔW_{min} threshold must be applied to reduce the false negative rate. The minimum weight based false negative shows the same result, thus we skip this discussion.

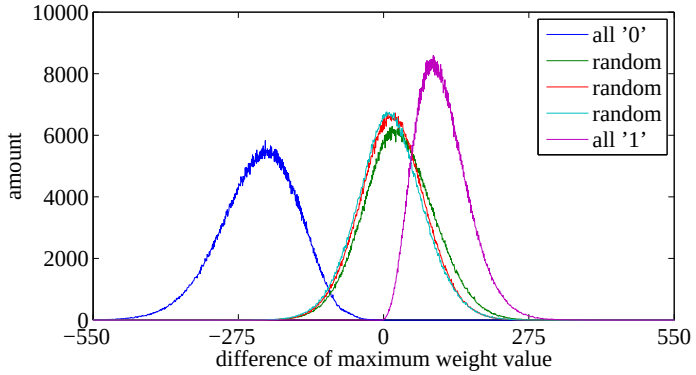

Fig. 5.12: ΔW distribution for certain responses with random 12 bit errors

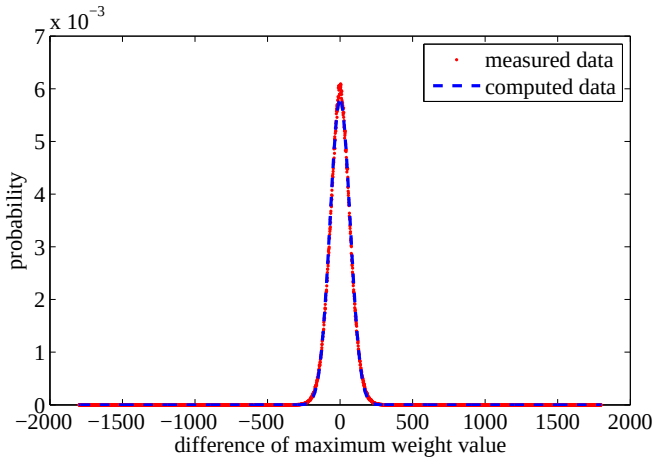

Fig. 5.13: Tolerant capabilities of the maximum weight difference value

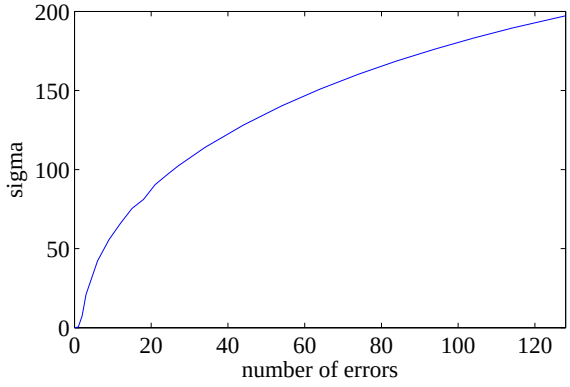

Fig. 5.14: Relationship between error number and σ

Location-based False Negative

As with the weight-based fault-tolerance scheme, determining the location-based false negative rate is also a hard problem that cannot be solved in the polynomial time. Therefore, we provide the statistical results in Figure 5.15 instead. The results contain seven groups of simulations with different errors in responses. Again, the number of errors follows a normal distribution. In each group, ten millions authentication processes with

different response patterns are simulated. We calculate the average values to evaluate the probability of $|\Delta L_{max}|$. To simplify the problem, we use the absolute values of the location difference. With a limited interval from -15 to 16, the results are more easily affected by the errors. Thus, the curve shows different features compared to the normal distribution. Meanwhile, the shapes of the curves are also determined by the weight distribution in the ring. If the weight values in the ring are given in a linear way, the curve shows a continuous feature. Otherwise, it is stepped. Compared to the curve of 4 bit errors, groups with more errors have higher level of randomness in distribution. Clearly, when there are 64 bits flipped in a 128-bit response, it cannot keep its own feature any more. Therefore, the curve with 64 bit errors is as flat as an uniform distribution.

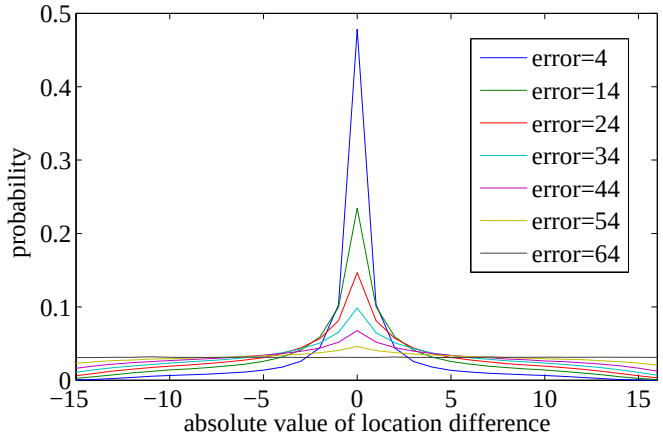

Fig. 5.15: Relationship between error number and $|\Delta L_{max}|$

One way to improve the performance of the algorithm is to use logic operations of both location-tolerance and weight-tolerance. The intersection of location and weight

set can decrease false negatives but increase false positives. Contrarily, the union of location and weight set decreases the false positive rate but increases the false negative rate. The combination of the peak and valley values plays similar function. As the error rate grows, wider ranges of location and weight value are necessary to keep a low false negative. However, a larger set of values sacrifices the false positive for tolerating more bit errors. In a case that each 128-bit response of PUFs contains 12 random errors, a pure fault tolerant algorithm with very limited overhead cannot keep both false positive and false negative very low in theory. Thus, while false negative is required to be as low as possible, how to avoid the increasing of false positive becomes a critical issue.

5.3.3 Location Shift Scheme

Since fault tolerance is involved in the algorithm, false positive becomes an ineluctable problem for consideration. To decrease false positives, a location shift scheme is proposed. This scheme requires an ID set of at least 32 bits in unauthenticated chips. As shown in Figure 5.16, the low 20 bits record a serial number, which is the basic address of the fault tolerant array. Since we simulate one million chips, it requires at least 20 bits to generate unique IDs. The high 8 bits represent a random number that relates to the offset of the location. The middle 4 bits are encryption bits for data transfer security consideration. When a chip receives a challenge from authenticating system, the 32-bit ID will not be sent to the system until the response is generated. To transfer data securely, the 128-bit response and 32-bit ID are mixed together logically. Certain bits of

the basic address are flipped according to different 4-bit code word patterns. The 8-bit offset address is also encrypted with an 8-bit challenge by XOR operation. Specially, the eight bits of the challenge are selected by the quotient of the serial number division. When the 160-bit data are received by the authentication system, encrypted basic address, code word, encrypted offset address, and response are extracted according to our defined protocol. After receiving the response, ring weight module generates four values: peak weight, peak location, valley weight, and valley location. The locations are further shifted by offset address in order to extend the matching space to the original location value plus the offset value. Meanwhile, four values calculated in advance in the fault tolerant array are selected by the decrypted basic address and are sent to the comparator for authentication.

By adding the offset values, the original locations are extended from 32 to 256. Therefore, it is harder for a cloned chip with a faked ID to point to the correct location. Since the collision probability of locations is reduced, the false positive decreases as a consequence. In Figure 5.16, the offset values added to the peak and valley location are different because the valley location shift method uses revered offset bits, which can further increase the difficulty of the collisions. Another effective way to decrease the false positive is to extend the offset address. When the random bits in the ID are 11 bits, the location shift space is up to 2048. The probability of false positive can be written as

$$P_{fp}^{s} = \frac{|\Delta L|}{s} \tag{5.11}$$

The variable s is the location shift space. The original false positive is reduced significantly with a set of s between 256 and 2048. Hence, increasing the shift space makes random responses of a cloned PUF harder to find the correct peak and valley location and pass the authentication test. However, expanding the shift space destroys the ring architecture of ΔL. The original ring is cut and extended to a line, which means the ± 16 becomes ± 32. The distribution of location-tolerance will not follow the theoretical equations any more.

5.3.4 Security Consideration

Apart from the accuracy rate, security is another important issue for authentication. The underlying PUF is still susceptible to potential modeling/ML attacks and RWA is not designed to address such issues in the PUF. Moreover, commercially it is not feasible for attackers to build many models since our focus is on authenticating a large amount of chips. However, an attacker could potentially generate a response that matches the RWA weight or location values. If we only use one simple RWA, it is not difficult to get two random responses with the same value. RWA has two thresholds related to collisions: $t_{maxlocation}$ and $t_{maxweight}$. The maximum location of RWA leads to a uniform distribution, as all the results lies evenly in 32 buckets. Here we apply the birthday paradox to evaluate the difficulty in finding a response that passes the maximum location threshold. As adapted from an argument of Paul Halmos, the probability that no two ΔL_{max} coincide is:

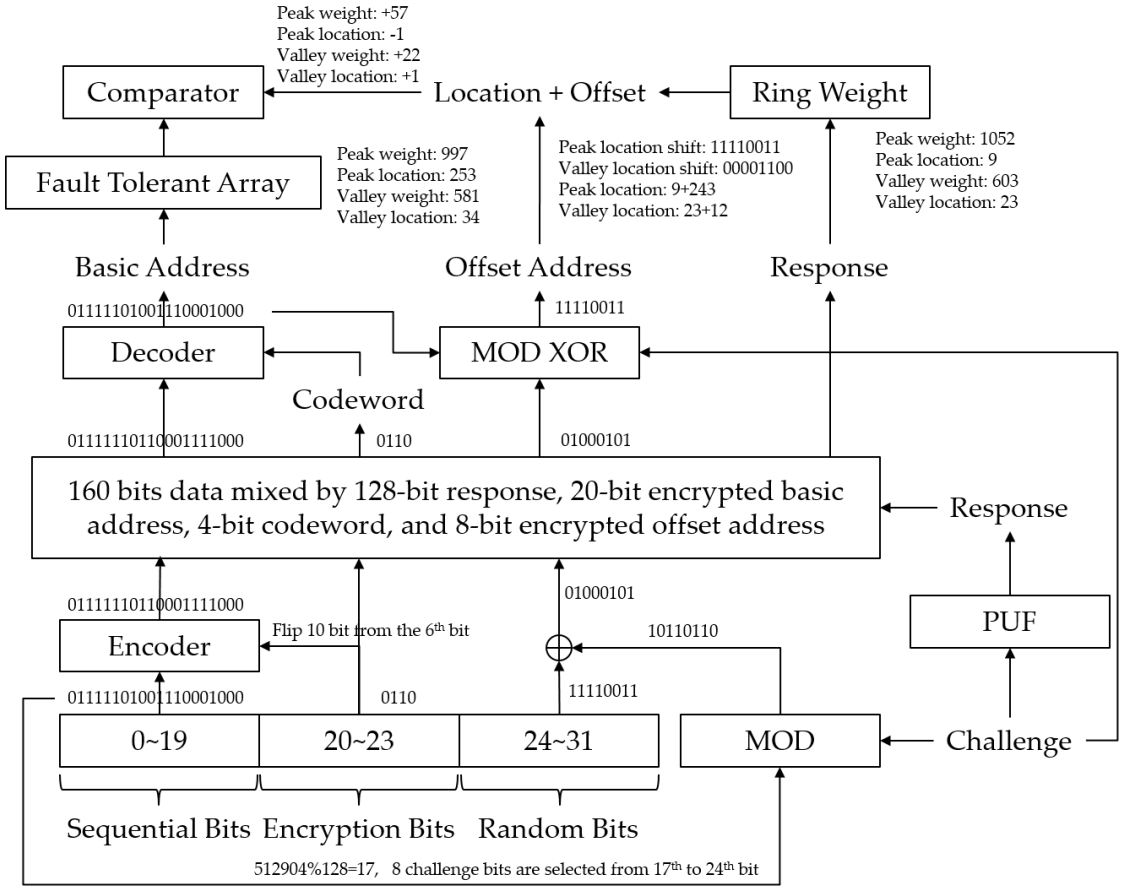

Fig. 5.16: Location shift scheme diagram

$$\overline{p}(n) = \prod_{k=1}^{n-1}(1 - \frac{k}{32}) < \prod_{k=1}^{n-1}(e^{-k/32}) = e^{-(n(n-1))/(2\times32)} \tag{5.12}$$

The above expression shows the interest in which the lower bound of a collision occurs when $p(n) > 1/2$, thus the inequality implies $\overline{p}(n) < 1/2$. Solving for n gives:

$$n^2 - n > 2 \times 32 \ln 2 \tag{5.13}$$

This inequality is satisfied when $n = 8$. Therefore, 8 responses suffice for ensuring that two hashes match with equal chance. Meanwhile, we assume that an upper bound is reached when $p(n) > 0.999$. Solving $n^2 - n > 2 \times 32 \ln 1000$ gives an approximate upper bound of $n = 22$. A very higher rate of finding a collision pair can be a potential security problem. By expanding the location range from 32 to 2048, the bound limitations are changed to 54 and 169.

The maximum weight of RWA, however, does not follow a uniform distribution. As shown in Figure 5.3, the maximum weight follows a normal distribution $N(1008, 106^2)$. Assuming p_x and p_y are probability density functions for the maximum weight of two random responses, the average probability that two hash values coincide is:

$$p_{average} = \sum_{x=y=0}^{1800} p_x p_y = 0.0027 \tag{5.14}$$

This means that one pair of collisions may be found in about 371 pairs of responses. Though the maximum weight should theoretically match the range of weight from 0 to 1800, it barely appears at both sides of the normal distribution, but is centralized near the mean value of the curve. By replacing 32 with 371 in Equation 6.9, the lower bound and upper bound of weight-based collision are given as 24 and 73.

Mixed RWA refers to a combination of 11-bit maximum weight value, 11-bit minimum weight value, 5-bit maximum location value, and 5-bit minimum location

value. The theoretical maximum possibility of the mixed RWA value is the multiplication of four parts ($370.37 \times 370.37 \times 32 \times 32$), By taking four factors together, the upper bound and lower bound increase significantly. Table 5.2 shows the number of responses an attacker must try to achieve a 50% or 99.9% probability of success. Compared to the ECC/Bloom filter approach, our mixed RWA scheme is much more difficult for attackers to generate fitting responses and find a collision. The number of guesses can be limited to enhance the security level.

Table 5.2: Security Comparison of Different Schemes - Number of response guesses required to achieve 50% or 99.9% success rate

scheme	50%	99.9%
ECC & Bloom filter	128	402
maximum or minimum location-based RWA	8	22
maximum or minimum location shift RWA	54	169
maximum or minimum weight-based RWA	24	73
mixed RWA	13955	44053

5.4 Evaluation and Discussion

In this section, results of different authentication schemes are compared: no ECC, traditional ECC only, ECC plus Bloom Filter, Ring Weight Algorithm, Helper Data Algorithm, and fuzzy extractor. We provide simulation results of authenticating one

million chips with different schemes. To measure the collision rate, another group of one million cloned chips are simulated with random responses. Since it is not practical to implement one million different real PUFs, we use simulation and assume the PUF responses have robust and uniform bit strings according to [11, 20, 42]. The PUF bit probabilities are independent and there is no significant bias in the PUF responses. Environmental conditions and aging only generate bit errors, but do not change our assumptions. In our simulations, we assume a roughly 10% bit error rate such that the bit errors in each response follow a normal distribution around a mean of 12. These assumptions track with results obtained in the literature [1, 11, 49]. Thus, the simulated results can present the real case of PUFs. Apart from simulation results, we also test the performance of RWA based on the implementation of ring oscillators-based PUFs (RO PUFs) and arbiter PUFs on Xilinx FPGA.

5.4.1 Error Correcting/Bloom Filter Matching

We start with a 16×8 ECC coding where the 128-bit response is split up into eight groups, each of which has its own BCH code word. Thus, one response with an average of 12 errors will be decoded eight times before authentication. The false positives and false negatives are shown in Table 5.3. m, t, k and r refer to the dimension of Galois Field, number of errors that can be corrected, length of information bit, and number of parity checks. BF counter defines the number of chips that cannot be authenticated. ECC counter presents the number of responses that cannot be corrected by the BCH

code. For the best case, only three chips (out of one million) failed to be correctly identified through the Bloom Filter and error correction. Data rate refers to the efficiency of the error correction scheme - i.e. $\frac{k}{k+r}$, the ratio of number of data bits to number of bits in the code. As can be seen, the overhead for these methods is quite high - to achieve the low error rate, one requires an overhead of 48 bits or three times the data.

Table 5.3: 16×8 Bits ECC/BF Authentication Results

m	t	k	r	data rate	BF counter	ECC counter	false positive	false negative
6	4	16	24	0.40	42556	42588	2.54×10^{-4}	4.26×10^{-2}
6	5	16	28	0.36	4731	4734	2.54×10^{-4}	4.73×10^{-3}
6	6	16	36	0.31	356	356	2.54×10^{-4}	3.56×10^{-4}
6	7	16	44	0.27	20	20	2.54×10^{-4}	2.00×10^{-5}
6	8	16	48	0.25	3	3	2.54×10^{-4}	3.00×10^{-6}

The overhead can be improved somewhat with a 128×1 BCH coding solution as shown in Table 5.4 at the cost of higher false negative rates. However, it still has a nearly 100% overhead in memory space. Also, the system is much more compute intensive due to a larger amount of encoding and decoding calculation.

5.4.2 Weight and Location Tolerance

To simplify the bit error correction processing, the ring weight algorithm is composed of the following parts: peak weight value, peak location value, valley weight value,

Table 5.4: 128×1 Bits ECC/BF Authentication Results

m	t	k	r	data rate	BF counter	ECC counter	false positive	false negative
8	12	128	92	0.58	226059	226124	2.54×10^{-4}	2.26×10^{-1}
8	13	128	100	0.56	83739	83771	2.54×10^{-4}	8.37×10^{-2}
8	14	128	108	0.54	16512	16522	2.54×10^{-4}	1.65×10^{-2}
8	15	128	116	0.52	4072	4076	2.54×10^{-4}	4.08×10^{-3}

and valley location value which are calculated according to the responses. If the value locates in the tolerance range, then the response passes the authentication. Figures 5.17 and 5.18 show the performance of different location and weight tolerance ranges with 12-bit errors in each 128-bit response. For most cases, the simulated false positive and false negative rates match the theoretical result. With wider tolerance ranges, the false negative decreases but the false positive increases.

When using this authentication method, the difference of maximum weight and its probability of false positive should follow a normal distribution. The standard deviation σ depends on the range of weight value. A wider range of weight makes σ larger, which can filter more false responses for the authentication process. The false positive probability related to ΔW is given by

$$P_{fp}^{\Delta W} = \sum_{i=0}^{k} \binom{n}{i} p^i (1-p)^{n-i}, p = p_\phi \tag{5.15}$$

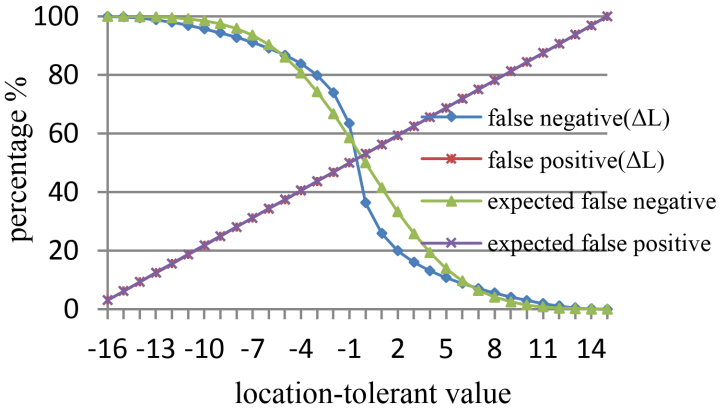

Fig. 5.17: False negative and false positive rate related to location-tolerance

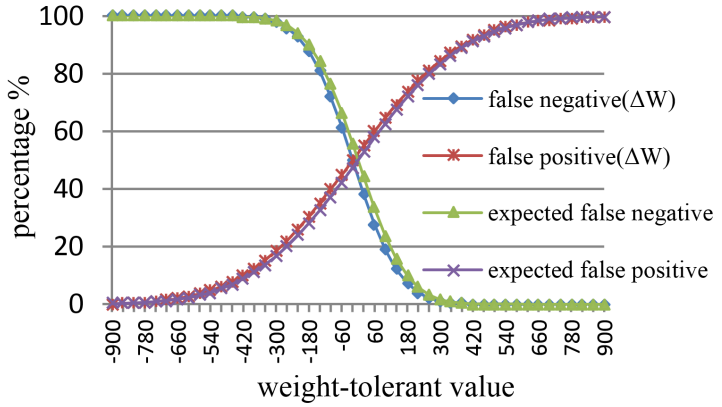

Fig. 5.18: False negative and false positive rate related to weight-tolerance

5.4.3 Location Shift

The effect of the location shift scheme is shown in Figure 5.19. The offset address bits are extended to 11 bits in order to obtain up to 2048 location shift. As can be seen, the false positives drop significantly as peak and valley location value are shifted by offset bits.

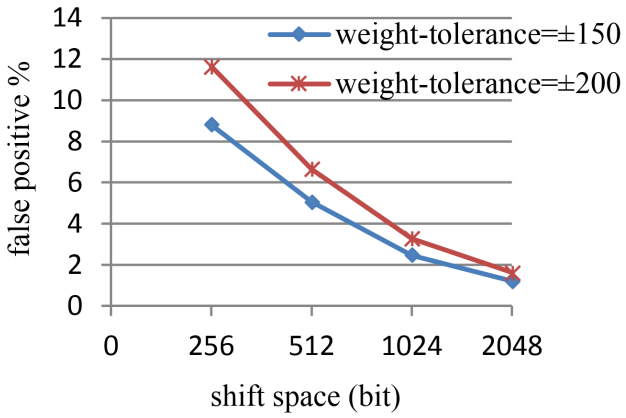

Fig. 5.19: False positive of different shift spaces

5.4.4 Fault Tolerance Capacity

To measure the fault tolerant capacity, the average bit errors in responses is set from 3 to 18. Since bit error does not affect the false positive rate, which refers to a random response with or without bit errors, we only focus on the relationship between the bit error number, tolerance range, and the false negative rate. In Figures 5.20 and 5.21, $[-3, 3]$ location-tolerant range and $[-70, 70]$ weight-tolerant range show distinct shifting, which can help reduce the false negative rate but increase the false positive rate. When the tolerance value continues increasing, the data come to a convergence.

5.4.5 Weight Optimization

To further improve the authentication accuracy, the deviation of weight set $\{w_1, \ldots, w_{32}\}$ is reduced. Giving up the weight trade-off showed in Figure 5.2, we merge the values to meet the requirement of false negative. Various weight patterns are measured, among

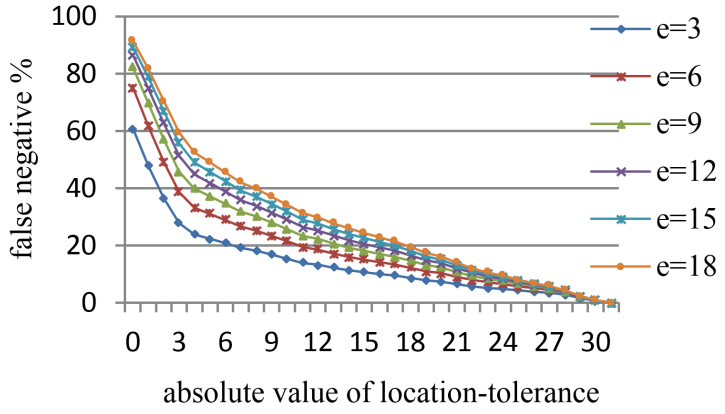

Fig. 5.20: False negative of different bit errors with different location-tolerance

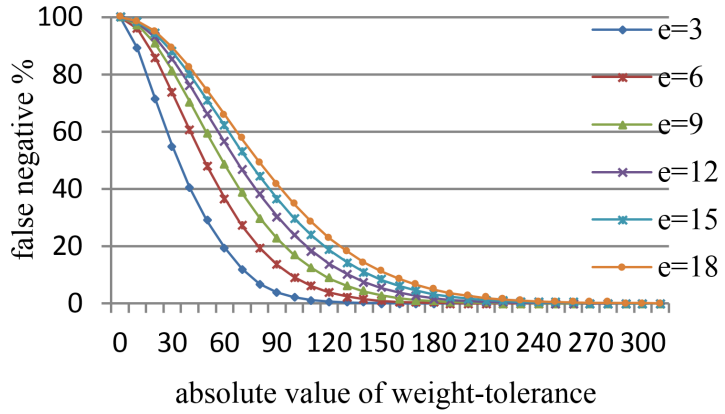

Fig. 5.21: False negative of different bit errors with different weight-tolerance

which results of five groups are shown in Figure 5.22. By reducing the maximum weight pattern from 16 to 7, the false negative decreases from 1.58% to 0.02%.

5.4.6 Offset Bit Flipping

For the location shift scheme, reversing the offset address bit before shifting the valley location value is an efficient way to further reduce false positives. Since the peak and

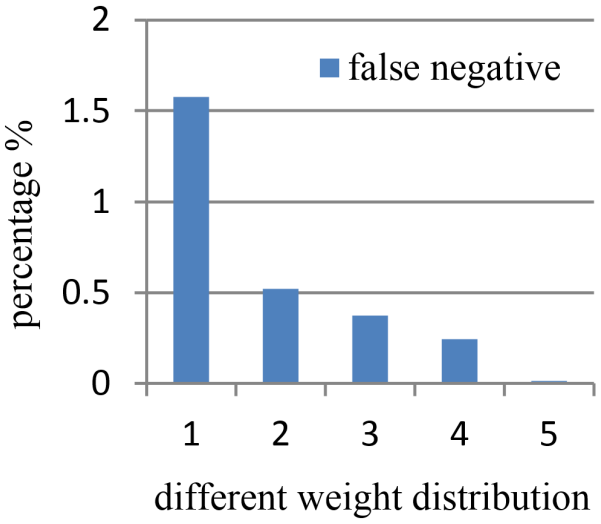

Fig. 5.22: False negative of different weight distribution methods

valley location values are added to different shift values, the probability of getting a collision becomes smaller. In Figure 5.23, 150*s* and 200*s* refer to same offset bits with ±150 and ±200 weight-tolerant range. 150*d* and 200*d* add different location shift values for original peak and valley location. As a result, offset bit flipping reduces the false positive rate greatly.

5.4.7 Application Field

RWA can be applied to different types of PUFs. However, depending on the particular PUF implementation, the distribution of 1's and 0's in the PUF response may not be uniformly distributed or may even have a bias to '1' or '0'. Both the weight-tolerance and location-tolerance approaches have some resistance to these variations. We simulated nine groups of CRP authentications and show the worst cases in Figure 5.24.

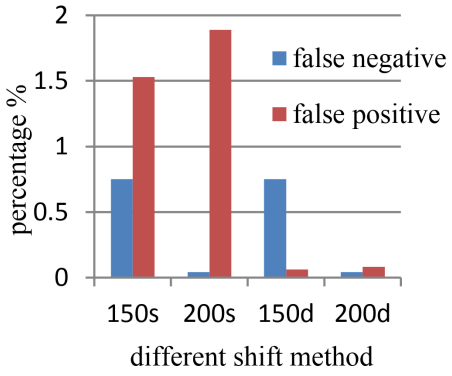

Fig. 5.23: False positive and false negative rates of different 2048 location shift methods

With a 20% bias from the ideal baseline of 50% '1' and 50% '0', the false negative rate is still less than 0.5%. However, if 0's or 1's reach significant majority, any minority occurrence due to the bit error will obtain larger chance to cause the shift of the overall weight and location value. The false positive rate depends on both the bias in the baseline response as well as the bias in the response under test. The *false positive 1* curve shows the false positive rate if both responses have the same bias, and we can see that the false positive rate remains constant at less than 10^{-3}. The curve of *false positive 2* shows a different situation, under which the bias of the baseline varies from 30% to 70% while the response under test keeps a bias of 50%. As a result, the false positive is reduced when there are more differences between the response pair, which means a collision will be more easily found with the same bias. Therefore, if the response of a PUF shows good randomness and acceptable bias, RWA can archive the lowest false negative. The bias will only help to reduce the false positive when the bits between the

baseline and the response under test have more than 5% difference.

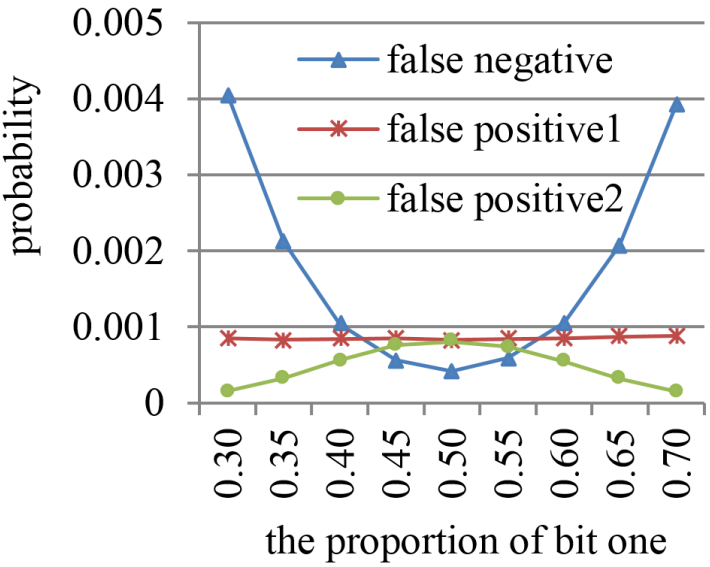

Fig. 5.24: Effect of response bias on false positive and negative rates

5.4.8 Overall Performance Consideration

Figure 5.25 shows the effect of adding various optimization methods to RWA. We start with baseline and then add the location shift optimization, weight optimization and offset bit flipping optimization. By involving these schemes step by step, we can reduce the false positive and false negative significantly with only a slight increase in overhead. Figure 5.26 shows how the number of errors affects the false negative rate. The effect becomes significant with more than 6 errors per 128 bits. Since the false positive rate is not affected by the number of errors, but is a consequence of the compression due to RWA, the false positive rate curve remains flat.

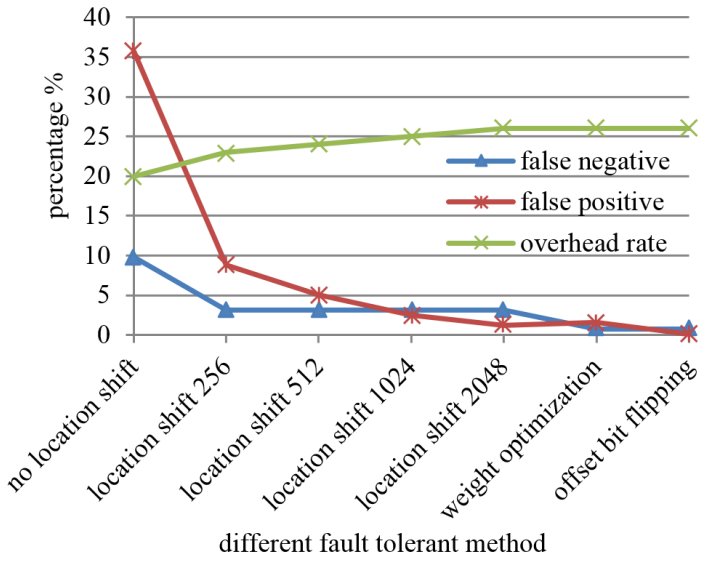

Fig. 5.25: Performance comparison of different fault tolerant methods

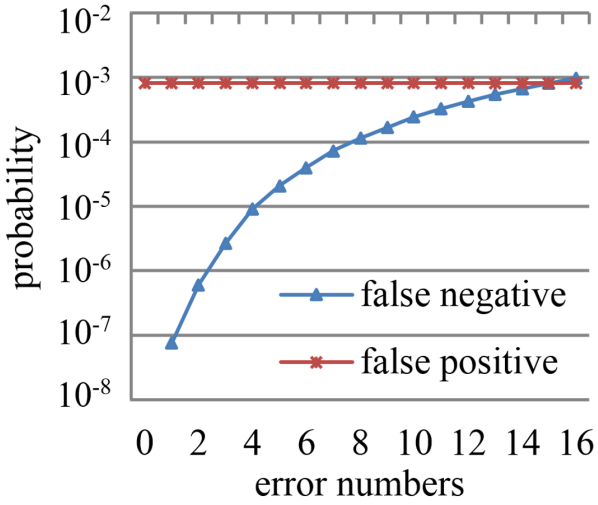

Fig. 5.26: Overall effect of number of errors

Table 5.5 shows the overall comparison of the different solutions. Since the bit error rate follows a normal distribution, the probability that no error occurs in 128 bits becomes infinitesimal. If no ECC is used for direct bit matching, no chip passes authen-

tication. Once BCH ECC is applied, the false negative decreases while the overhead increases. The ECC and Bloom Filter solution provides acceptable false positive and false negative rates at the cost of large computation and memory space. Our novel ring weight algorithm and corresponding optimization shows accurate authentication rate without ECC. We achieve the best performance by setting the absolute value of location-tolerance to 4 firstly. When the difference of two peak locations and the difference of two valley locations are smaller than 4, the response under test is authenticated as passed. If it is not satisfied, we extend the absolute value of location range to 32, but set the absolute value of weight-tolerance to 200. Given a response with its baseline that cannot meet all the constraints of the peak weight, valley weight, peak location, and valley location, it will be regarded as failed. Smaller memory space and less computation are also its advantages compared to other solutions. Meanwhile, it does not cause security issues since the collision is hard to find, even if attackers copy the location shift information.

There has been some prior work to use non-ECC helper data to reduce failure rates - in particular the Helper Data Algorithm (HDA) [8] and the fuzzy extractor [9]. In Table 5.6, we show a comparison of these algorithms with our RWA algorithm. The reported error rates, memory overhead, and failure rates come from the relevant papers. For our RWA algorithm, we show a range of error rates and the respective failure rates. As can be seen, HDA produces 128-bit PUF responses with a failure rate of 10^{-6}. However, the memory overhead of HDA is 317 times that of RWA. If a nonce

Table 5.5: Performance Comparison of Our Solutions

solution	memory space (MB)	average 10 errors		average 12 errors	
		false positive	false negative	false positive	false negative
No ECC	15.26	0	1	0	1
ECC only	38.15	0	1.71×10^{-2}	0	4.26×10^{-2}
ECC & BF	15.85	2.54×10^{-4}	9.88×10^{-4}	2.54×10^{-4}	1.65×10^{-2}
Ring Weight	4.53	8.81×10^{-2}	5.18×10^{-2}	8.81×10^{-2}	6.41×10^{-2}
Ring Weight (Opt.)	5.25	8.20×10^{-4}	1.66×10^{-4}	8.20×10^{-4}	4.28×10^{-4}

is applied to the CRPs, the memory space taken by HDA would be a heavy burden to a database. Moreover, RWA can authenticate one million CRPs in 9 seconds on average while HDA takes 206 seconds. The fuzzy extractor algorithm needs 450 ring oscillators to build one 128-bit response with a failure rate of 10^{-6}. As the bit error rate increases, more ring oscillators are required to keep a low failure rate. However, RWA does not rely on hardware. In addition, though not reported in [9], the calculation complexity of the fuzzy extractor is much higher than that of RWA. With much smaller memory space, RWA can be applied to any type of PUF. Therefore, HDA and the fuzzy extractor have good applications in PUF-based key generation where reduced failure rate is important. However, the large size and computational complexity, make it impractical for large-scale chip authentication where a database would need to store multiple challenges for each device to be authenticated. Conversely, RWA exhibits the best trade-off and can be an acceptable PUF post-processing solution for IC authentication.

Table 5.6: Overall Comparison with Other Solutions

	HDA	fuzzy extractor		RWA
PUF type	SRAM PUF	RO PUF		any type
error rate	15%	9%	21%	9% to 21%
helper data memory overhead (bit)	13952	450	770	44
PUF bits required for 128 bits	1536	450	770	128
failure rate	10^{-6}	10^{-6}		10^{-5} to 10^{-3}
computing time (μs per response)	205	not mentioned		9

5.4.9 Performance of RWA with PUF Implementations

To take the measurement noise, bit bias, and other factors into consideration, we evaluated the performance of RWA with real CRPs from ring oscillator (RO) PUFs [1] and arbiter PUFs [39]. We implemented three RO PUFs on a Kintex-7 FPGA and applied 128-bit challenges to generate 128-bit responses. All the FPGA LUT-based inverters in the ROs are well-placed to guarantee the equivalent delays between any selected pair. Every challenge, which is generated by a pseudo random number generator, is measured 20 times on each PUF. The first 10 responses are used to generate a baseline while the average value of the last 10 responses is regarded as an unauthenticated response under test. As a strong PUF, each PUF provides 100,000 challenge-response pairs. Table 5.7 shows the response bias, bit error rate, false negative rate and false positive rate. As we mentioned in Figure 5.24, *false positive 1* is computed between a baseline and a random response with the same bias while *false positive 2* uses a random response with a fixed 50% bias. Note, the computed value of false negative from RO

PUF 3 is shown as 0, but it should be regarded as less than 1.00×10^{-5} as we only authenticate 100,000 groups of data. We also evaluated the use of arbiter PUFs, which are also implemented on Kintex-7 FPGA. Though all the MUX cells are well-placed, the responses of arbiter PUFs still have more than 5% bit bias. Compared with the simulation results in Figure 5.24 and Figure 5.26, the measurement errors of both false negative rates and false positive rates are acceptable. These results show that RWA is effective in authenticating real PUFs.

Table 5.7: Performance of RWA with Real PUFs

PUF type	bias	bit error rate	false negative	false positive 1	false positive 2
RO PUF 1	54.68%	11.56%	1.09×10^{-3}	8.40×10^{-4}	8.90×10^{-4}
RO PUF 2	49.76%	7.13%	4.80×10^{-4}	8.00×10^{-4}	9.00×10^{-4}
RO PUF 3	51.06%	3.12%	$< 1.00 \times 10^{-5}$	8.30×10^{-4}	9.10×10^{-4}
arbiter PUF 1	41.26%	3.31%	$< 1.00 \times 10^{-5}$	6.40×10^{-4}	5.80×10^{-4}
arbiter PUF 2	44.71%	4.22%	6.00×10^{-5}	8.10×10^{-4}	9.10×10^{-4}
arbiter PUF 3	41.52%	2.89%	3.00×10^{-5}	7.10×10^{-4}	7.00×10^{-4}

If we were willing to tolerate large numbers of bit errors in the responses, one could conceivably improve the false negative rates significantly. The following analysis provides an estimate of bounds on the false negative rates with errors. We collected one million responses from RO PUF 1, and unlike the NO ECC solution in Table 5.5, this

time, a threshold is set as the fault tolerance method. In other words, if the compared responses have fewer unmatched bits than the threshold, the authentication regards the response as a success. Since the bit errors in RO PUF 1 follow a normal distribution, a few responses with 20 or more errors require a large threshold. As shown in Table 5.8, we need to tolerate more than 1/4 fault bits of the response. As a result, even if we set no constraints on the number of errors and storage space, the performance can be hardly improved. Applying ECC or other helper data schemes to improve the performance increases the storage needs significantly which makes them less practical in industry settings. On the other hand, our RWA approach achieves comparable failure rates with reduced storage than this simple threshold-based scheme or other more advanced schemes.

Table 5.8: False Negative of Bit-by-bit Comparison

threshold	false negative	threshold	false negative
0 bit	1	20 bits	8.65×10^{-2}
4 bits	9.98×10^{-1}	24 bits	1.30×10^{-2}
8 bits	9.47×10^{-1}	28 bits	1.06×10^{-3}
12 bits	7.01×10^{-1}	32 bits	4.30×10^{-5}
16 bits	3.26×10^{-1}	36 bits	1.00×10^{-6}

5.5 Conclusions

In this chapter, three solutions are presented for industrial authenticating challenge-response pairs from physically unclonable functions. Our simulation, based on one million PUFs, shows that a practical error correcting and Bloom filter matching scheme achieves 10^{-4} false positive rate, 10^{-2} false negative rate, and a data rate of 0.58. Our proposed fault tolerant ring weight scheme is implemented using three rings which indicate weight, location and response values. With different types of optimization, it can achieve a false positive rate of 10^{-4} and a false negative rate of 10^{-4} with a data rate of 0.74. We also apply true CRPs from RO PUFs and arbiter PUFs to show that RWA is effective in practice. Moreover, the new scheme, which does not rely on ECC, requires little computing time and memory space. The lower bound and upper bound of collisions are acceptable in the case of real industry needs. For a given amount of storage space, our scheme provides the best authentication accuracy.

Chapter 6

PUF Initialization Table for Robust Authentication and Key Generation

6.1 Introduction

In the previous chapter, we proposed a novel PUF design. Though the bit error rate has been reduced significantly, post-processing is still required for many applications. In this chapter, we propose a PUF initialization Process (PIP) to record the entire feature of a PUF, which can improve the stability in different applications. To evaluate the practicality, we provide two PIP-based authentication solutions with very low false positive and false negative rate. Additionally, we combine the error correction and bit selection scheme together by proposing a PIP-based key generation solution using our floating thresholding algorithm and helper data, which improves the stability of responses for various PUFs.

6.2 PUF Initialization Process

When PUF applications require reproducible responses with the same challenges, storing CRPs is a direct, high-cost and unsecure storage method. If we take into account the instability feature, which needs a strong ECC for post-processing, the whole solution becomes more time-consuming and memory-wasting. In this section, we provide a simple but practical data structure to record the useful features of PUFs, which is regarded as a fundamental element of improving the bit selection efficiency and PUF stability.

6.2.1 PUF Initialization Table Generation

The PUF initialization table (PIT) is a critical part in our solution, as the basic idea of PIP is to test the stability of all the RO pairs combinations in a PUF in order to generate a PIT with trusted output values. To create the PIT, we use an initialization challenge generator (*ICG*) to feed our optimized RO PUF with repeated challenges. Algorithm 4 states the details of PIP: We first fix the higher address register *addr_A* and change the lower address register *addr_B* from 0 to 127. After 1,000 repeated measurements, the register values of *CounterL*, *CounterS* , and *CounterE* are recorded in the PIT, in which we set the base address to *addr_A*, and the offset address ranges from 0 to 127. Then the test will restart between the next RO in array A and the 128 ROs in array B. When the PIP is completed, the PIT contains an entire feature copy of a PUF, which can be used for authentication and key generation. It should be noted that other PUFs do not

need to record *CounterE*, as they are unnecessary to generate output by comparing the counter values.

6.2.2 Analysis of PIT Model

When we downloaded the same bitstream file into the same FPGA, the generated PIT had minor changes. A solution is to calculate the average value of each parameter among multiple PITs from the same PUF. Here we use 10 samples to build the model. This PIT model presents more accurate features of a PUF compared to a random PIT. Data analysis of the PIT model is given in Table 6.1. Since all 16,384 RO pairs are tested 1,000 times, the sum of the three states is 1,000 for any selected challenge address. Each state has a counter value ranging from 0 to 1,000. In our 3-state PUF, *larger*, *smaller*, and *equal* state have roughly equal percentages. Among the 16,384 RO pairs, there are more than 3,000 *larger* cases and *smaller* cases with a counter value of 1,000 - in other words all 1000 tests give the same *larger* or *smaller* results. There are no cases where the *equal* state was recorded in all 1,000 measurements. This indicates that $\frac{3105+3014}{16384} = 37.35\%$ RO pairs can generate very stable responses. We call them the confident set. Though it cannot guarantee the same values completely during the PIT regeneration, the stability is enough to identify a 128-bit response.

Algorithm 4 PUF initialization process

procedure PIP
    reset $CounterL$, $CounterS$, $CounterE$, i, $addr\_A$, $addr\_B$
loop 1:
    reset $CounterA$, $CounterB$
loop 2:
    read $RO(addr\_A), RO(addr\_B)$
    if $clockcycle < 16$ **then goto** *loop 2*
    if $CounterA > CounterB$ **then**
        $CounterL \leftarrow CounterL + 1$
    else if $CounterA < CounterB$ **then**
        $CounterS \leftarrow CounterS + 1$
    else if $CounterA = CounterB$ **then**
        $CounterE \leftarrow CounterE + 1$
    $i \leftarrow i + 1$
    if $i < 1000$ **then goto** *loop 1*
    output $CounterL, CounterS, CounterE$
    reset $CounterL, CounterS, CounterE$
    $i \leftarrow 0$
    $addr\_B \leftarrow addr\_B + 1$
    if $addr\_B < 128$ **then goto** *loop 1*
    $addr\_B \leftarrow 0$
    $addr\_A \leftarrow addr\_A + 1$
    if $addr\_A < 128$ **then goto** *loop 1*.

Table 6.1: Parameters of Our PIT Model

Counter A vs. Counter B	larger	smaller	equal
percentage of each state	33.93%	34.16%	31.91%
state counter value = 0	7384	7178	7838
state counter value = 1000	3105	3014	50

6.3 PIT-based Authentication Application

A direct application of PIT is PUF-based authentication, which is shown in Figure 6.1. Before embedding an RO PUF into a chip or a system, the PIP must be invoked once to obtain the unique PIT. The *ICG* becomes active during PIP and enables RO pairs in our predefined sequences. Meanwhile, *CounterL*, *CounterS*, and *CounterE* in the optimization unit collect the data from *CounterA* and *CounterB* in the RO PUF. These counter values are sent to FIFO and will be transferred to a database as a PIT. When the PIP finishes, normal CRP tests are launched for authentication. Ideally, our CRPs will only select RO pairs that have been deemed stable by the PIP. However, since we use a *Challenge Shift Register* to select RO pairs, it requires every 14-bit output from the 128 shifts to select a stable RO pair and the probability of finding such a challenge is only 37.35%. In order to address this problem, we provide block RAM-based solution and threshold-based solution.

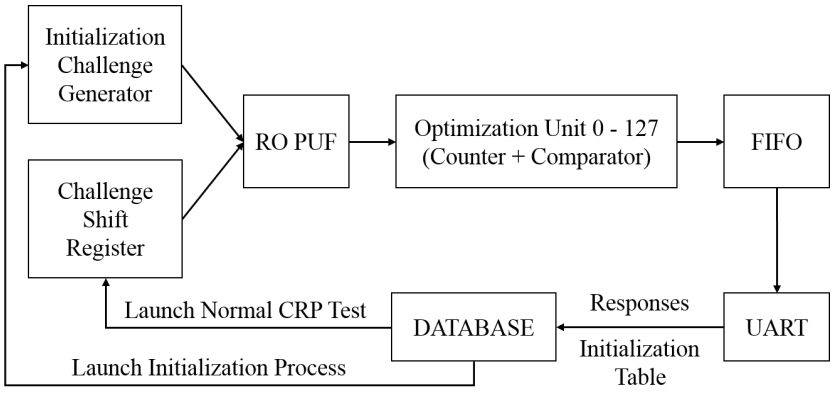

Fig. 6.1: Our PUF-based authentication design using PIT

6.3.1 Block RAM-based solution

One solution is to replace the challenge shift register with a block RAM (BRAM), which has a width of 14 bits and a depth of 128. Since there is no constraint on the input pattern, a challenge can be generated in the database by selecting 128 addresses from the confident set randomly. Then the 14×128 bits challenge is sent to the block RAM in the FPGA. The counter values are read as responses for authentication. Table 6.2 gives a typical protocol of the communication and the authentication. Since the challenges will only select stable RO pairs, a counter value of 1,000 means 100% stable. Other than that, the probability of the counter value follows half of the normal distribution $N(1000, 1)$ from 0 to 1,000 according to our tests. That is to say, when the counter value is located as 999, 998, and less than 998, it roughly stands for a confidence interval of 68%, 27%, and 5% in the normal distribution. Therefore we can simplify the response by using 3-bit coding instead of transferring counter values directly. The highest bit

stands for the response output. The lower two bits present the stability of the highest bit. Every value is multiplied with their corresponding score during authentication. To decide whether a CRP passes or fails, the sum of these multiplication results is compared with a threshold score. We find a balance between false positive and false negative by testing different scores and thresholds.

Table 6.2: Protocol of Our Authentication Method

range of counter value	tolerance interval	output	score
$CounterL = 1000$	μ	111	0
$CounterL \in [999, 1000)$	$[\mu - \sigma, \mu)$	110	1
$CounterL \in [998, 999)$	$[\mu - 2\sigma, \mu - \sigma)$	101	3
$CounterL \in [0, 998)$	$[0, \mu - 2\sigma)$	100	5
$CounterS = 1000$	μ	011	0
$CounterS \in [999, 1000)$	$[\mu - \sigma, \mu)$	010	1
$CounterS \in [998, 999)$	$[\mu - 2\sigma, \mu - \sigma)$	001	3
$CounterS \in [0, 998)$	$[0, \mu - 2\sigma)$	000	5

To prove the false negative bound, we compare the PIT model with five PITs generated by random measurements. The average occurrence times of score 1, 3, and 5 are marked as i_m, j_m, and k_m. In the random challenge generation process without selecting repeated RO pairs, a set (i, j, k) is chosen for authentication, in which $i \in [0, i_m]$, $j \in [0, j_m]$, $k \in [0, k_m]$. Among all the combinations of (i, j, k), some subsets

$(\hat{i}, \hat{j}, \hat{k})$ fail in the authentication. Therefore, the false negative rate is

$$f_n = \sum_{(\hat{i},\hat{j},\hat{k})} \frac{\binom{L+S-i_m-j_m-k_m}{128-\hat{i}-\hat{j}-\hat{k}}\binom{i_m}{\hat{i}}\binom{j_m}{\hat{j}}\binom{k_m}{\hat{k}}}{\binom{L+S}{128}} \tag{6.1}$$

The subsets also need to meet the following constraints:

$$\hat{i} + \hat{j} + \hat{k} \leq response\_bit\_width \tag{6.2}$$

$$\hat{i} + \hat{j} \times 3 + \hat{k} \times 5 > threshold\_score \tag{6.3}$$

The false positive proof is based on a scenario that the attacker knows our algorithm and only the 111 and 011 outputs are used to maximize the collision rate. Therefore, in the set (i, j, k), both i and j are 0. Collision occurs when no more than k_t values are mismatched in order to meet the constraint $k_t \times 5 \leq threshold\_score$. In that case, the false positive is

$$f_p = \sum_{k=0}^{k_t} \frac{1}{2^{128}} \binom{128}{k} \tag{6.4}$$

6.3.2 Threshold-based solution

In another solution, 128-bit challenges are generated randomly without referring to the values in a PIT so that we do not need to modify the *Challenge Shift Register* in the RO PUFs. Instead, the values of *CounterL*, *CounterS*, and *CounterE* are sent to the

database directly. This is because unstable RO pairs are included by random challenges. In order to tolerate the increasing instability, the authentication requires more accurate information. We also notice that the distributions of the counter values from unstable RO pairs are not as stable as those of stable RO pairs, which means using the confidence interval of a fixed normal distribution curve is impractical. Therefore, we propose an improved solution in Algorithm 5. We first check if the counter value of the *larger* state or *smaller* state has high stability in the corresponding address of the PIT model. Slight floating measurement errors are tolerated. When both states show low stability, we need to check all three states to evaluate the confidence degree. If no constraint is met, it is regarded as a failed RO pair. A CRP will pass authentication only when the failure number is smaller than a threshold t.

Algorithm 5 Segment of threshold-based authentication

1: **procedure** THRESHOLD
2: $score = 0$
3: **for** $i \in \{0, \ldots, 127\}$ **do**
4: get PIT $addr$ from the challenge
5: **if** $PIT(addr).CounterL > 800$ and $\Delta CounterL < 100$ **then** $n \leftarrow n$
6: **else if** $PIT(addr).CounterS > 800$ and $\Delta CounterS < 100$ **then** $n \leftarrow n$
7: **else if** $\Delta CounterL + \Delta CounterS + \Delta CounterE < 500$ **then** $n \leftarrow n$
8: **else** $n \leftarrow n + 1$
9: **if** $n \leq t$ **then** CRP passes

To calculate the false negative bound in this solution, we need to know the number of failed RO pairs between the PIT model and an unauthenticated PUF. Marked as m, the number ranging from $t+1$ to $min\{m, 128\}$ will cause an authentication failure. The false negative rate can be written as

$$f_n = \sum_{n=t+1}^{min\{m,128\}} \frac{\binom{16384-m}{128-n}\binom{m}{n}}{\binom{16384}{128}} \tag{6.5}$$

The false positive rate is similar to Equation 6.4, but the probability of meeting the constraint is not fixed to $\frac{1}{2}$. We use p to represent the probability of a random collision in 128 trials.

$$f_p = \sum_{n=0}^{t} \binom{128}{n} (1-p)^n p^{128-n} \tag{6.6}$$

Traditional PUF applications require significant storage space for CRPs. For the same architecture of our RO PUF, our PIT can generate $\binom{16384}{128}$ challenges given a fixed 128-bit data length. However, traditional authentication methods need almost infinite memory to store all the CRPs. By sticking to only stable challenges from the confident set, the memory size is reduced to $\binom{6119}{128} \times 2 \times 128 = 8.23 \times 10^{264}$ MB. It still takes 2.50 MB to store only 10,000 CRPs for one PUF. Meanwhile, a small CRP set is not a secure way for authentication. If all the CRPs in the memory are used, a replay attack will be 100% successful. However, our PIT solution only requires 480 KB memory size to generate all possible CRPs for authentication. All we need to store are the initialized counter values (30 bits) for 128×128 RO pairs. It can be even smaller (48 KB) if we

replace the counter value with a 3-bit threshold. Another advantage of using PIT is, with a set of 3.37×10^{268} stable responses, we can apply a cryptographic nonce to our challenge generation in order to prevent replay attack.

6.4 PIT-based Key Generation Solution

PIT provides sufficient accuracy for authentication purposes. However, PUF-based key generator is much more difficult to be implemented. Given the same input, PUFs are required to consistently produce exactly the same output, which has been proved to be almost impossible due to the instability. To eliminate the bit errors, error correction codes are widely applied as industrial solutions. But the hardware memory of a PUF cannot afford the huge overhead when a large amount of keys are generated. On the other side, bit selection schemes hold a strong capability of avoiding using the most unstable bits but fail to eliminate the remaining errors. With each advantage in mind, we propose a PIT-based key generator in this section by combining these two solutions together.

6.4.1 PIT-based Floating Thresholding Bit Selection Scheme

Instead of selecting the most stable bits only, which reduces the CRP entropies significantly, we focus on finding a floating threshold for the current PUF response to minimize the effect from the systematic variation and the environmental noise. Our target is to reach a balance among the stability, entropies, and resources. The proce-

dure begins with generating five PITs from the same PUF, as shown in Algorithm 6. Before each PIT generation, we launch an entire PUF reset except the processor and the memory in order to enhance the accuracy of the stability estimation. This is because the RO PUF responses measured after physical reset show poorer stability than those running continuously without poweroff according to our tests. All five PITs are stored in the memory temporarily. Then, we fetch the counter values of the *larger*, *smaller*, and *equal* states from the first RO pair of each PIT. These five groups of data can be presented as five points on a 3D surface, which are shown as the red marks in Figure 6.2. Meanwhile, the 16,384 blue marks provide a clear distribution of counter values from a randomly selected PIT. In the next step, the processor calculates the center and radius of an enclosing circle for the five points. As we set the PIT loop to 1,000 ($counter_{larger} + counter_{smaller} + counter_{equal} = 1,000$), all the points are enclosed in a circle instead of a sphere. The circle can be regarded as the possible range of counter values for the first RO pair. Since the distribution shows a convergence trend to two lines ($counter_{larger} + counter_{equal} = 1,000$ and $counter_{smaller} + counter_{equal} = 1,000$), we can simply consider the $counter_{larger}$ or $counter_{smaller}$ within the overlapped range between the circle and the two lines. After storing the overlapped boundary in the memory, we repeat the process for the remaining 16,383 groups of data and accumulate the new overlapped boundaries to the previous ones. Once completed, we obtain the possible location distributions of the *larger* state and *smaller* state for all the RO pairs in a PUF, which are shown as group 1 in Figure 6.3. The cumulative value of 0 is about

7,800 while the cumulative value of 1,000 is about 3,100. For the remaining part, we can find the minimum value and set the x-axis as a floating threshold. Particularly, the locations are 860 and 909 for the *larger* state and *smaller* state in group 1. Thus, we connect (860,0,140) and (0,909,91) with a magenta line in Figure 6.2 as the floating threshold of group 1. Now we select 128 points by hashing a challenge plus an offset. If a point is below the floating threshold and very close to either of the two blue lines, it generates a 1-bit response. Otherwise, it is regarded as unstable and will be ignored during the key generation. For either case, the challenge is incremented by 1 to generate a new hash value to select the next point.

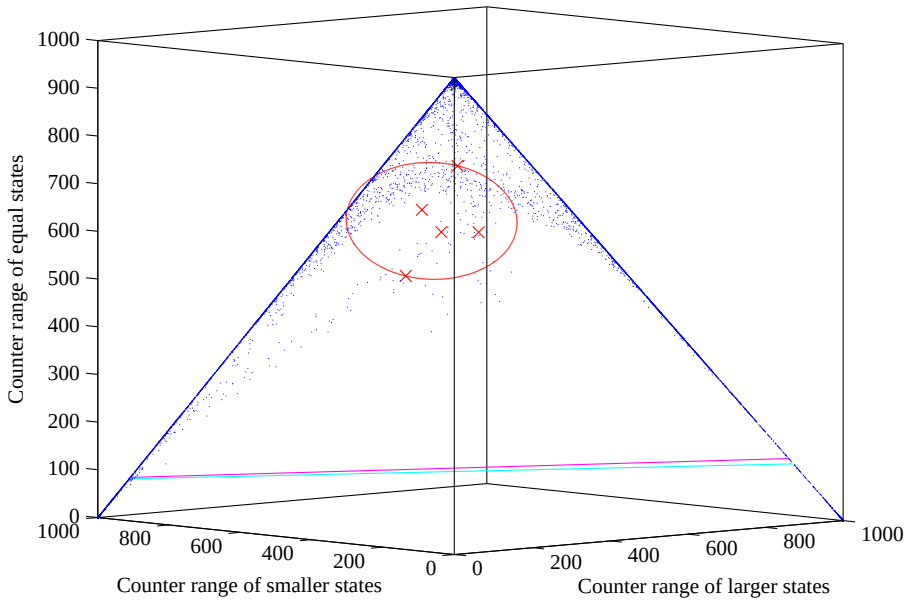

Fig. 6.2: 3D distributions of counter values and the floating threshold generation of three states for the RO PUF responses

Algorithm 6 Segment of floating thresholding solution

```
1:  procedure FLOATING THRESHOLDING
2:      receive Challenge
3:      for i ∈ {1, ..., 5} do
4:          reset PUF except MicroBlaze and SDRAM
5:          generate PIT(i) and store it in SDRAM
6:      for j ∈ {1, ..., 16384} do
7:          fetch RO pair(j) counter values from PIT(1, 5)
8:          calculate the enclosing circle of the five points
9:          cumulate and store the overlapped boundary
10:     if Helper Data is invalid then
11:         set the minimum boundaries as floating threshold
12:         select 128 stable points with hash(Challenge++)
13:         generate Helper Data for less stable RO pairs
14:         output Response and Helper Data
15:     else
16:         regenerate the floating threshold with new PITs
17:         decode Helper Data
18:         select 128 stable points with hash(Challenge++)
19:         output Response after error correction
```

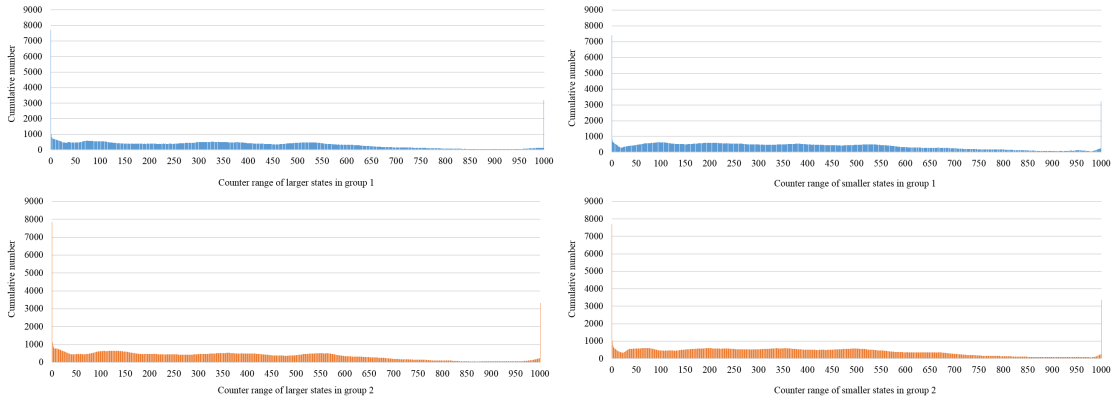

Fig. 6.3: Possible location distribution of larger states and smaller states based on two groups of RO PUF test results

After group 2 is generated in Figure 6.3, a new floating threshold is plotted in Figure 6.2 as the cyan line. In this case, the new floating threshold is very close to the original magenta line. If the counter position of an RO pair changes between the stable and unstable state easily, we define it as a marginal point. Among all 16,384 points, there are about five whose center points stay on different sides of the floating thresholding lines. In the worst case, we assume that any of these five RO pairs can cause a failure when they are selected. Then the upper bound of the failure rate in 128-bit keys is

$$p_{\ floating\ threshold} = 1 - \frac{\binom{7424}{128}}{\binom{7429}{128}} = 0.0833 \tag{6.7}$$

in which 7429 is the number of points that are located below the threshold line in group

1. On the other hand, if we use a fixed threshold, the failure rate is

$$p_{\ fixed\ threshold} = 1 - \frac{\binom{7524}{128}}{\binom{7622}{128}} = 0.8118 \tag{6.8}$$

6.4.2 PIT-based Floating Thresholding Error Correction Scheme

If challenges are generated by the PUF itself, we can avoid selecting the RO pairs with marginal points near the thresholds. Then, there is no extra overhead for error correcting. However, for the general case when challenges are provided from another device, helper data is required to correct any potential bit error in the responses. Unlike traditional ECCs, which address the entire binary string, we apply a low-cost helper data that only focuses on the sensitive bits generated by the points near the thresholds. Given the top five RO pairs with potential error correction requirements in our example, we rank them in order of stability. In Figure 6.4, they are located in the 46th, 116th, 47th, 24th, and 42nd bit in a 128-bit key circle with black buckets. To generate the helper data of the 46th bit, we set a start point at 0 and a distance. Now we search for another four locations with the equivalent distance between each, one of which must locate in 46, but none of which locates in 24, 42, 47, and 116. With a distance of 23, these five red buckets in the middle circle contain only one point near the floating threshold. Once a key is generated, we apply XOR operations to the bits located in Set 1 $\{0, 23, 46, 69, 92\}$, which will generate 1-bit codeword. To generate Set 2 and the codeword for the next bit located in 116, we shift the start point to 1 and repeat the

previous steps. Since the distance of 81 is too large to generate the remaining three locations, we can use the least positive remainder after divided by 128. Thus the bits in the location of $\{1, 82, 35, 116, 69\}$ generate the second codeword, which are shown as the orange buckets in the middle and external ring. Similarly, we select the green, blue, and purple buckets for the sets related to 47, 24, and 42.

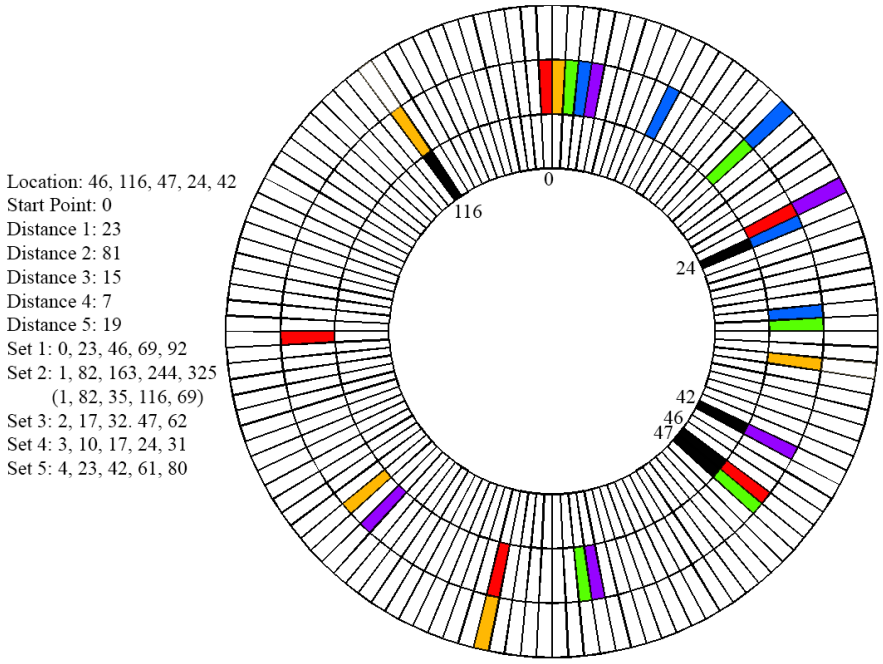

Fig. 6.4: Helper data generation for our floating thresholding key generation solution

Given the start point (s), locations (L_n), and distances (D_n), the helper data generation should follow

$$\exists! L_n = s+n-1+i\times D_n \;\; i \in [0, m-1] \forall L_{\overline{n}} \neq s+n-1+i\times D_n \;\; i \in [0, m-1] n \in [1, n_{max}] \quad \overline{n} \in [1, n_{max}] \quad n \neq \overline{n} \tag{6.9}$$

in which the n_{max} is the number of marginal points and m is the number of buckets selected for XOR operations. In our example, both n_{max} and m are set to 5. When a key is generated for the first time, the input helper data is all 0 and regarded as invalid. The helper data, including start point, locations, and parity, is generated and output with its response. During the regeneration, the same helper data is received along with the original challenge. While most incorrect RO pair selections are solved by the new floating threshold, the rest ones can be corrected by the decoded helper data. Comparing these two groups of data, few points may stay on one side of the first threshold but the other side of the second threshold. If a marginal point is selected to generate one bit and it stays on the same side of a new threshold during the key regeneration, it can be easily verified by the decoded locations. Once it merges to the other side, there is 1-bit right shift for the binary string after this point. With the information of the recorded marginal points, this error can be detected and corrected easily. Given a marginal point ignored by the original key, it can be aware by the decoded locations during the key regeneration unless there is a location collision. But we are still able to detect an error since the entire locations and the parities are not matched. To correct the potential 1-bit right shift, we can try to ignore the recorded marginal point and check the new response with the helper data. In order to minimize the collision rate, we apply an additional checksum to the original key. For example, the checksum can be a 10-bit parities or hash value generated by the entire key. If there are several bit strings that match the locations and

the parities from the decoded helper data, we can compare the checksum of the new key with the original one to find out which one should be the correct key.

The exploding memory utilization of PUF-based key generation is more critical than other applications because of the highest accuracy requirement. Traditional solutions usually apply BCH ECC to correct as many as t bit errors in 128-bit responses. If a typical RO PUF has a bit error rate of 9% in average, and the error number distribution requires an ECC with 18-bit correction capability. The number of parity-check digits r can be calculated as $r = n - k = 2^m - 1 - 128 \leq mt = 18t$. The smallest t that meets the inequality is 8, which means r is at least 127 bits. According to our measurement, the failure rate is higher than 10^{-2}. To archive a lower failure rate, the overhead can be much larger than the generated key. The linear expanding storage requirement limits the application. For the floating thresholding solution with a key length of 128 bits, the start point is $log_2 128$ bits. Since the distances are variable, we fix the length using the largest one D_{max}. D_{max} is recommended to be set to 127 according to our simulation, as it is large enough to find all the locations. The codeword also includes n_{max}-bit parities and c-bit checksum. Thus, the entire codeword has $7 + log_2 D_{max} \times n_{max} + n_{max} + c$ bits.

6.4.3 Security Consideration of PIT-based solutions

Apart from the increasing cost of resources, PUF applications also suffer from being attacked or modeled as the PUF information stored in the database can be easily exposed to adversaries. For our key generation solution, this vulnerability can be safely ignored

as a PIT will be deleted after generating sufficient pairs of challenges and helper data. Unlike ECC, the remaining data will not disclose any information that could be used to recover the relevant responses.

With respect to PUF components, most modeling attacks are based on the assumption that the PUF structure is known [17]. Given k ring oscillators in the RO PUF, two of them are selected and compared in order to produce a single output bit. This structure leads to $k(k-1)/2$ CRPs. For the worst scenario, the CRP interface is public and accessible. Then one can measure all the RO frequencies $(f_1, ..., f_k)$ in ascending order. The approach requires a low-degree polynomial calculation with a complexity of $O(k \cdot logk)$. However, it is inefficient and expensive to open the package, probe the circuit, and apply the attack in reality. First of all, one needs to identify the PUF type, which is difficult on an FPGA. Secondly, the interface may not release the original information of CRPs. When a hash function is added between the challenge and the RO input, the learning complexity increases. XOR architecture can also be applied between the RO output and the response, which is proved to be effective against the machine learning attacks [17]. If we take these facts into account, the 128-bit keys in our solution are still secure against attacks with limited cost.

6.5 Data Analysis and Evaluation

6.5.1 Experimental Setup

In this section, we continue to use RO PUFs to evaluate our solution step by step, with results measured from three Kintex-7 FPGA based KC705 boards. Note that each FPGA contains six PUFs. The temperature on the FPGA is controlled at 30°C and the voltage is 1.0V. To evaluate the stability, we also use temperature variation instrument to test the PUF with different temperature ranges.

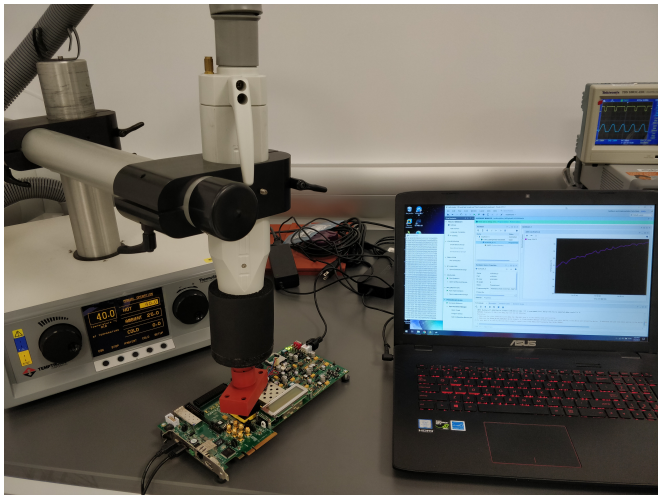

Fig. 6.5: PUF tests with temperature variation instrument

6.5.2 Hardware Resource Utilization

Here, we estimate the hardware cost of PIT-base PUF. The overall utilization on the Kintex-7 is only 1.75% of the available look-up tables and 0.65% of the flip-flops, which is shown in Table 6.3.

Table 6.3: FPGA Resource Usage of One PUF Implementation

ref name	used	functional category
FDRE	2640	Flop & Latch
LUT6	1718	LUT
LUT1	1026	LUT
MUXF7	598	MuxFx
LUT5	346	LUT
LUT3	203	LUT
LUT2	191	LUT
MUXF8	189	MuxFx
OBUF	132	IO
LUT4	94	LUT
FDCE	20	Flop & Latch
LDCE	14	Flop & Latch
IBUF	4	IO
CARRY4	4	CarryLogic
FDSE	1	Flop & Latch
BUFG	1	Clock

6.5.3 Randomness

The NIST statistical test suite is a common technique used for entropy source and cryptographic analysis [50]. Each input file contains 100,000 128-bit responses from one PUF. All 18 PUFs with PIP pass the 15 NIST randomness tests, with cumulative

p-values larger than 0.975.

6.5.4 Uniqueness

The uniqueness of the 18 PUFs is shown in Figure 6.6. The comparison is based on the same PUF placement and bitstream in the three FPGAs. Thus, all three PUFs in each group have no circuit level differences. We compare 100,000 128-bit responses from each PUF with the same challenge set. All six groups have 50% ± 0.97% bits (64 bits) flipped on average. The results are also verified by comparing the values of their PIT models.

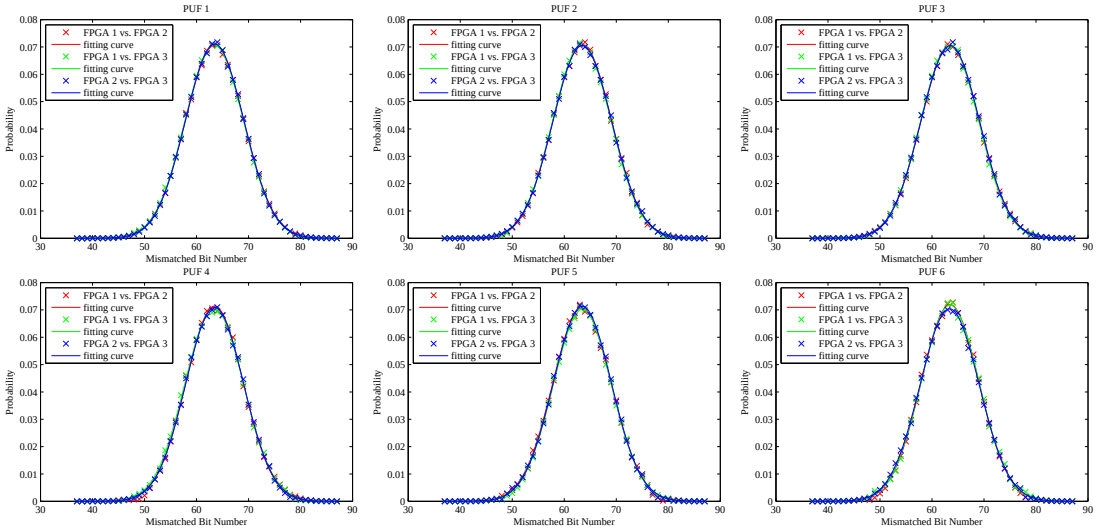

Fig. 6.6: Inter-PUF hamming distances

6.5.5 Stability

Figure 6.7 provides the stability comparison among the traditional design, lightweight optimized design, and RO PUF with PIP. The raw bit error rate (RBER) in the traditional RO PUF is 7.13% in average. A few responses contain more than 20 bit errors, which necessitates strong fault tolerance or error correction method. With our optimization, the bit error rate decreases significantly. Further, if we only select the stable RO pairs for output according to the PIT model, 76.73% of the responses have no bit errors. Within the temperature range of a normal testing environment, the observed bit error rates of our PUFs with PIP are less than 0.50%.

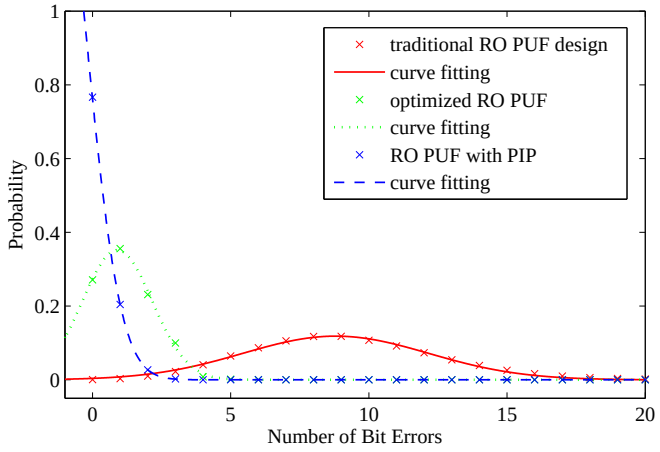

Fig. 6.7: Intra-PUF hamming distance

Table 6.4 lists the average bit error rates from 18 PUFs. Again, we mark the six PUFs on the second FPGA as $PUF6$ to $PUF12$ and those on the third FPGA as $PUF13$ to $PUF18$. Clearly, the stability of PUFs with PIP has been improved greatly compared

to the traditional RO PUFs.

Table 6.4: Average Bit Error Rate

PUF 1	PUF 2	PUF 3	PUF 4	PUF 5	PUF 6
0.33%	0.47%	0.43%	0.37%	0.40%	0.35%
PUF 7	PUF 8	PUF 9	PUF 10	PUF 11	PUF 12
0.31%	0.32%	0.29%	0.36%	0.37%	0.30%
PUF 13	PUF 14	PUF 15	PUF 16	PUF 17	PUF 18
0.34%	0.43%	0.45%	0.41%	0.31%	0.38%

6.5.6 Bias

The response bias to '1' in the traditional RO PUF design has an offset from −9% to 8% according to our tests - i.e. the PUF produces 41% to 58% 1's instead of the ideal 50%. Apart from the physical uncontrollability, the close frequency of RO pairs intensifies instability. Using our PIT architecture reduces the bias significantly to 50% ± 1.82%. There is a slight bias towards '0' in our PITs as shown in Figure 6.8. For example, in the PIT model of PUF 1, there are 3105 0's and 3014 1's, which means the bias of a random response is 50% − 0.73%. To achieve a bias of 50%, we can select the value of '0' or '1' in the PIT model when generating challenges. Clearly, the bias is well-controlled in our design.

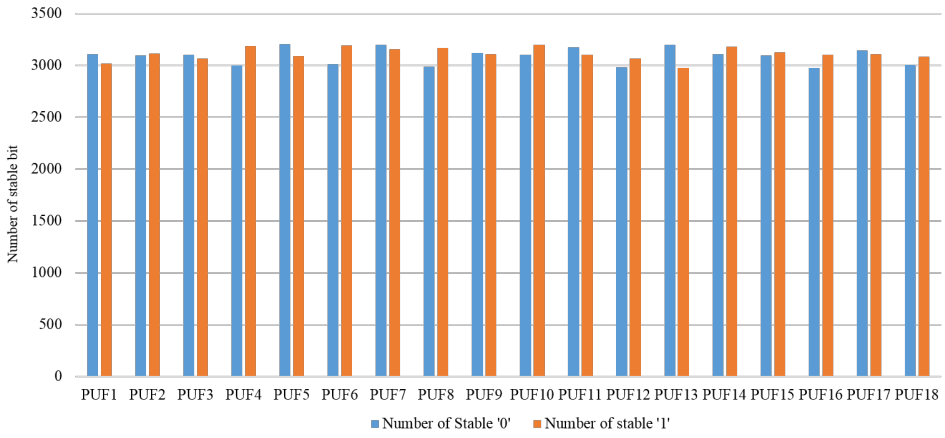

Fig. 6.8: Bias of 18 PUFs on three FPGAs

6.5.7 Result of PIT-based Authentication

The performance of our authentication solutions is tested on our RO PUFs with PIP. In the BRAM-based solution, the threshold score is set to to 39 in order to balance the failure rate and the collision rate. The results from six PUFs are given as an example in Table 6.5. There are few mismatched values among all the stable RO pairs. On average, only 29 combinations of RO pairs lead to authentication failures. Using Equations 6.1 and 6.4, we can calculate the false negative and false positive rates and we see that they are smaller than 10^{-26}. In the threshold-based solution, m is set to 68, the maximum number of failed RO pairs in any PUF. If we set the threshold, t, to 30, using equations 6.5 and 6.7, the false negative and false positive rates are smaller than 10^{-47}. Specially, attacks on the first solution do not reply on the PIT model, so the bound in the BRAM-based false positive rates are the same for all PUFs. However, the collisions in the threshold-based solution is related to the PIT model, which leads to different bound

in threshold-based false positive rates. Finally, we verified both solutions by testing 10 million random CRPs and no failure is detected during the authentication.

Table 6.5: Results of Authentication with PIP

	PUF 1	PUF 2	PUF 3	PUF 4	PUF 5	PUF 6
stable RO pairs	6119	6302	6262	6183	6291	6201
BRAM-based Solution						
(i_m, j_m, k_m)	(17,9,1)	(34,5,0)	(22,7,1)	(28,10,0)	(20,6,2)	(25,3,2)
$(\hat{i}, \hat{j}, \hat{k})$	29	22	23	70	23	7
false negative	3.62×10^{-27}	2.08×10^{-45}	2.91×10^{-33}	4.46×10^{-28}	3.53×10^{-30}	1.25×10^{-41}
false positive	2.95×10^{-28}	2.95×10^{-28}	2.95×10^{-28}	2.95×10^{-28}	2.95×10^{-28}	2.95×10^{-28}
threshold-based Solution						
m	53	49	52	68	60	47
p	0.1315	0.1137	0.1182	0.1250	0.1206	0.1093
false negative	3.79×10^{-53}	9.68×10^{-55}	1.58×10^{-53}	1.65×10^{-48}	9.00×10^{-51}	1.27×10^{-55}
false positive	1.06×10^{-59}	1.25×10^{-65}	4.82×10^{-64}	9.22×10^{-62}	3.19×10^{-63}	3.03×10^{-67}

6.5.8 Result of PIT-based Key Generation

Seven key generation HDAs are compared in Table 6.6. According to our tests, the failure rate of the traditional RO PUF is not good enough even after we apply a non-standard BCH(244, 128, 15) ECC. Both [8] and [9] have a failure rate of 10^{-6}, but they require large amounts of helper data. In [7], BCH(63, 24, 7) ECC is used to reach a failure rate of 10^{-38}. However, the data rate and key length are still limitations. As a comparison, our floating thresholding solution with helper data has quite a low failure

rate. In addition, the key length is variable while the other methods have fixed key. The helper data for each key requires only $7 + 7 \times n_{max} + n_{max} + 10$ bits, which means 7-bit start point, 7-bit location for each marginal point, n_{max}-bit parity, and 10-bit checksum. For more than 90% keys with floating thresholds, helper data is not even necessary. Therefore, the average data rate is very low.

Table 6.6: Overall Comparison of Error Correction and Bit Selection Helper Data Algorithms

solution	traditional RO PUF without ECC	traditional RO PUF with BCH ECC	HDA [8]	fuzzy extractor [9]	Samsung PUF [7]	floating thresholding without helper data	floating thresholding with helper data
PUF type	RO PUF	RO PUF	SRAM PUF	RO PUF	hybrid	RO PUF	RO PUF
key length (bit)	128	128	128	128	24	128 (variable)	128 (variable)
PUF cells	128	128	1536	450	not applied	256	256
helper data (bit)	0	116	13952	1850	39	0	$17 + 8 \times n_{max}$
data rate	1	0.52	0.01	0.06	0.38	1	≈0.98
RBER	7.13%	7.13%	15%	9%	30%	7.13%	7.13%
failure rate	1	4.08×10^{-3}	10^{-6}	10^{-6}	2.01×10^{-38}	8.72×10^{-2}	9.38×10^{-8}

We also evaluate our solution on different types of PUFs, as shown in Table 6.7. To build a PIT with the same depth of 16384, the minimum number of basic cells for each PUF is given for comparison. As we mentioned, two 128-RO arrays are sufficient to construct the RO PUF PIT by selecting one RO from each array (128^2). For Arbiter PUFs and BR PUFs, their entropies benefit from the serial architectures, which require only 14-bit challenge input and related units to build a PIT (2^{14}). Thus, the basic cell amount should be as least 14×4 MUXes for 2-XOR Arbiter PUFs and 14×1 MUX and

DEMUX for BR PUFs. However, the response bits of Butterfly PUFs are independent as the selected cross-coupled latches do not affect the output of others. As a result, 16384×2 latches are used in the Butterfly PUF. We also provide the LUT and FF utilization for implementing a PIT for each PUF. To obtain the failure rate, we randomly generate one billion keys for each PIT with the same seed. For the Butterfly PUF that keeps a similar RBER as the RO PUF, its failure rate is slightly reduced, but still on the order of 10^{-8}. On the other hand, when the RBER is cut in half as with the Arbiter PUF, the failure rate decreases sharply to 10^{-9}. If the RBER further drops to the degree as that of the Bistable Ring PUFs, no errors were detected in the key set, which means the false negative is smaller than 10^{-9}. In summary, the performance of this algorithm is most affected by the raw bit error rate in the PUF responses, but its accuracy is sufficient for key generation on different PUFs.

6.6 Conclusions

In this chapter, we proposed a strong solution for PUF authentication and key generation. We investigated the possibility of PUF post-processing methods. We demonstrated the accuracy of PUF responses with initialization tables, which achieves an average bit error rate below 0.48%. By using PIT, the failure rate and the collision rate of authentication also decreased significantly. For the most critical key generation applications, we use PITs and dynamic floating thresholds to select the stable bits as output. With our helper data correction, the failure rates of four different PUF-based

Table 6.7: Performance Comparison of Floating Thresholding Solution with Helper Data on Four Different Types of PUFs

PUF type	RO PUF	Arbiter PUF	Butterfly PUF	BR PUF
key length	128	128	128	128
basic cells	256 ROs	56 MUXes	32768 Latches	28 DE/MUXes
LUT	3316	97	38960	75
FF	2675	68	16912	30
RBER	7.13%	3.31%	6.80%	1.42%
failure rate	9.30×10^{-8}	5.00×10^{-9}	8.50×10^{-8}	$< 1.00 \times 10^{-9}$

key generators are reduced to 10^{-8}. Finally, we evaluate the overall performance of the hybrid solution. The overhead is extremely low compared to the existing algorithms.

Chapter 7

Locality Sensitive Hash Function for Similarity Search and Clustering

7.1 Introduction

In an *e*-world, the rapid growth of data leads to the impossibility of searching large amounts of data by brute force, which leads to an ever-increasing need for fast indexing and searching of large databases. To make large-scale search practical, researchers have explored multiple approximate similarity search techniques. One novel and interesting approach is called the locality sensitive hash function, which solves the approximate or exact near neighbor search problem in high dimensional spaces. LSH hashes input items so that similar items map to the same "buckets" with high probability. Unlike conventional cryptographic hash functions, LSH aims to maximize the probability of a "collision" for similar items.

7.2 Locality Sensitive Hash Function Background

To define a formal definition of a function h in the general LSH family F [51], it should satisfy the following conditions for any two points p, q, which belong to a metric space (M, d): if $d(p, q) \leq R$, then the probability that $h(p) = h(q)$ is at least P_1; if $d(p, q) \geq cR$, then the probability that $h(p) = h(q)$ is at most P_2. A family is interesting when $P_1 > P_2$. Then F is called $(R, cR, P_1, P_2) - sensitive$ [52]. The easiest LSH software approach is bit sampling for Hamming distance, which selects a random bit from the input d-dimensional vectors: $\{0, 1\}^d \rightarrow \{0, 1\}$. It has the following parameters: $P_1 = 1 - R/d, P_2 = 1 - cR/d$. In spite of easy computation, bit sampling is not a suitable solution for the high bit error rate scenario, as it causes significant false positive and false negative [53]. Another popular LSH is MinHash [54]. By applying the Jaccard index, MinHash is defined as the size of the intersection divided by the size of the union of the sample sets. Thus, $P_r[h(A) = h(B)] = J(A, B) = |A \bigcap B|/|A \bigcup B|$. However, it has the same shortage as that of bit sampling. Given 128-bit input strings with 12 different bits, the identification rate is only 0.83, which is not enough for similarity search applications. To achieve better performance, TLSH is proposed to process a byte string by using a sliding window to populate an array of bucket counts and construct the digest body with different binary values [55]. However, it requires an input byte stream with a minimum length of 512 bytes. As a consequence, TLSH cannot be applied to strings with variable length.

7.3 Motivations

As we have mentioned in the previous chapters, FPGA-based PUFs are suitable for authentication and key generation. We would like to extend the similar architecture to a wider range of applications. To meet the essential requirement of identifying similar items, we propose a novel FPGA-based LSH (FLSH) design in this chapter. This new LSH is designed with strong fault tolerant capability and high identification accuracy in mind. The second advantage is that, since our approach uses asynchronous logic in FPGAs instead of the complex processor calculation, it is low-cost and fast compared to the software based LSHs. On the other side, FLSH can be applied to clustering as well. However, to achieve these goals, the implementation on FPGAs requires strict placement and routing in order to control the internal wire delay. Therefore, we explore the FLSH implementation details with complex placement and routing policies. To prove the reality of FLSH-based similarity search and clustering applications, we evaluate the physical characteristics of FLSHs, including randomness, convergence, and fault tolerant capability.

7.4 FLSH Design and Implementation

We present the details of our FLSH design and implementation in this section. Unlike the general software-based LSHs, FLSH does not rely on processors for hash calculation. Instead, it is mainly constructed with lookup tables (LUTs) and latches. This low-cost design is more flexible and can be easily integrated into other hardware system

designs.

7.4.1 LUT-based FLSH Array Design

FLSH is built around a 2D array comprised of $M \times M$ cells. The array scale depends on the available hardware resources and the input length. In our example, we choose an architecture that contains 16×16 cells, which can be implemented in 256 configurable logic blocks (CLBs) of a Kintex-7 FPGA. There are two types of cells: $(M-2)^2$ cells in the middle are connected with their neighboring cells by four bidirectional signals from different directions; and $4M-4$ cells on the boundary receive the inputs from the internal cells as well as a $4M$-bit input data. Figure 7.1 shows the interconnect between 9 cells in a 16×16 array. Each cell takes in the input data and generates a 4-bit hash output that is connected to its neighbors. Because of the asynchronous cycles in the array, it takes time for the outputs to converge to a stable output which is then saved in a $4M^2$-bit FLSH output register. Theoretically, each of the four input wires to a cell should have identical delays. However, due to the physical randomness in the manufacturing process, each FPGA will generate a unique hardware hash function under this architecture. As a summary of the hashing process, in our 16×16 array, when a input is ready to broadcast, the system enables the 256-cell array for automatic calculation until most of the bits converge. Then it sends the 1,024 bitmap to the FLSH output register.

Figure 7.2 shows the internal structure of an FLSH cell. The cell input $in[3:0]$

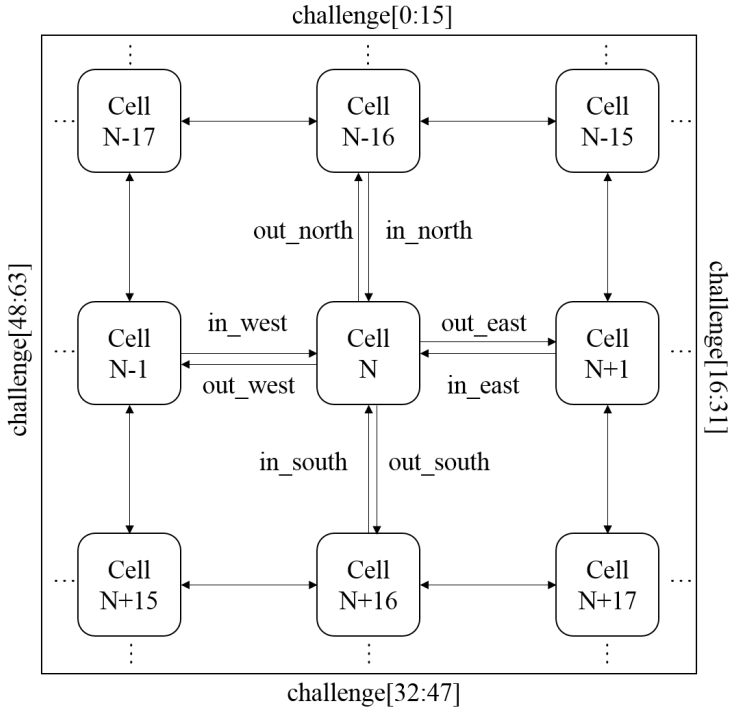

Fig. 7.1: Topology of FLSH network

comes from the neighboring cell outputs. It is connected to the main logic LUTs, latch LUTs, and shift LUTs in the cell. The logic LUTs ($LUT6\ NORTH$, $LUT6\ WEST$, $LUT6\ SOUTH$, $LUT6\ EAST$) determine the 4-bit output value of a cell. Given the difference of wire delay from the neighboring cells, we use $LUT4\ LATCH1$ to lock the logic LUT output at a certain time through its LDCE (Transparent Data Latch with Asynchronous Clear and Gate Enable). These outputs (*out_north*, *out_west*, *out_south*, *out_east*) are not only the input of the neighboring cells, but also part of the FLSH output. As we know, a 4-bit input can select a 1-bit output from the 16-bit initial values in a LUT. However, according to our tests, if we only use $in[3:0]$ as logic

LUT input, the output pattern is too simple in term of a hash function. Thus, shift functions ($LUT6\ SHIFT1$, $LUT6\ SHIFT2$, $LDCE\ SHIFT1$, $LDCE\ SHIFT2$) are added to improve the output complexity. Both $LUT6\ SHIFT1$ and $LUT6\ SHIFT2$ share the input from $in[3:0]$ and their LDCEs. For further timing consideration, we add another latch $LUT4\ LATCH2$ to lock $LDCE\ SHIFT1$ and $LDCE\ SHIFT2$ output. When the logic LUTs receive the 2-bit shift instruction from the LDCEs, there are four combinations of logic operations. We show an example in Table 7.1. If the shift bits are "00", the output will be selected from the original 16-bit initial value. When the shift bits change to "01", we apply a 1-bit left shift to the original initial value. When they are "10", a 1-bit right shift is executed instead. For the last case "11", all the 16 bits are flipped to generate new initial values. In this way, we generate the 64-bit initial lookup table for the logic LUTs.

7.4.2 FLSH Placement

Place and route is an essential step to control the wire delay in our FLSH in order to meet the critical timing requirement of LSHs. For each cell, the input signals should keep the same delay in order to involve the physical randomness of FPGAs themselves. To clarify this feature, the same challenge can generate unique hash values on different FPGAs, but one FPGA will always provide identical responses to very similar challenges. Instead of using timing constraints to achieve this goal, it is more reasonable to find some CLBs with equivalent delay path between each other. We select 256 CLBs

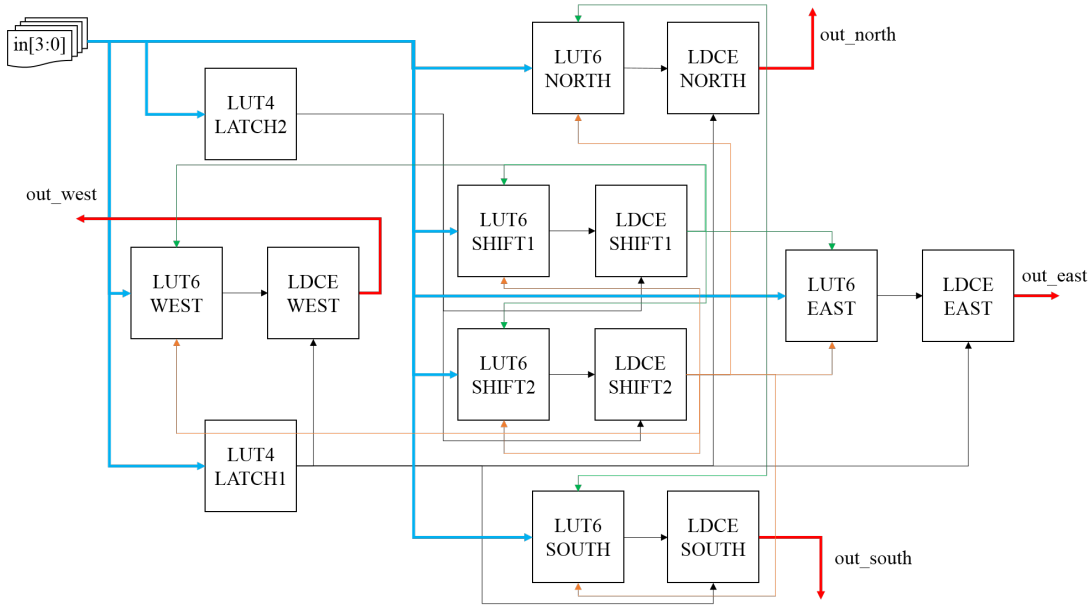

Fig. 7.2: Basic FLSH cell structure in a CLB

to build a 16×16 array in Figure 7.3. Note that there is a visible distortion that the distance in the middle looks larger than others, but the actual paths between those CLBs are very close to each other and can be adjusted by manual routing. Since the scale of the array may be changed according to the input length, manual placement is not efficient. To make the implementation flow more flexible, scripts are used to generate the placement constraints automatically.

According to our design requirement, a CLB should perform the entire function of an FLSH cell. In Kintex-7 FPGAs, a CLB provides two slices. In each slice, there are four 6-input LUTs and eight flip-flops as well as multiplexers and arithmetic carry logic [31]. Therefore, the primitives in a CLB is sufficient to build one FLSH cell. We

Table 7.1: Logic LUT Output of A Random FLSH Cell

neighboring cell input	output with shift0 = 0 shift1 = 0	output with shift0 = 1 shift1 = 0	output with shift0 = 0 shift1 = 1	output with shift0 = 1 shift1 = 1
0000	0	1	0	1
0001	0	0	1	1
0010	1	0	0	0
0011	0	1	0	1
0100	0	0	1	1
0101	1	0	0	0
0110	0	1	1	1
0111	1	0	0	0
1000	0	1	1	1
1001	1	0	1	0
1010	1	1	0	0
1011	0	1	1	1
1100	1	0	0	0
1101	0	1	0	1
1110	0	0	1	1
1111	1	0	0	0

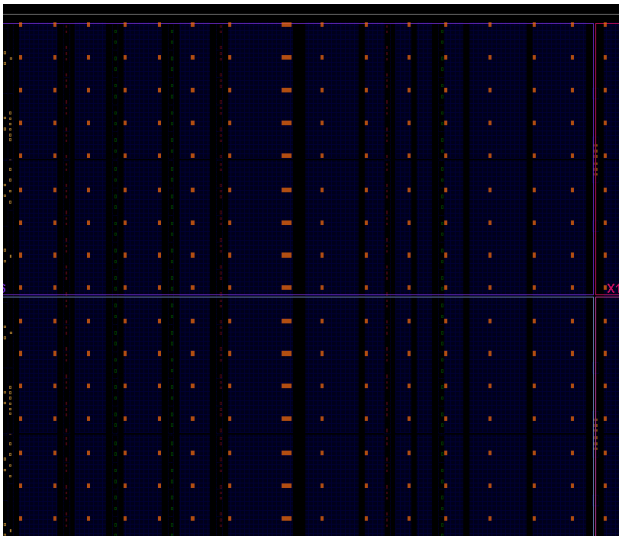

Fig. 7.3: FLSH array with 16 × 16 cells

provide an example in Figure 7.4, which shows how the LUTs and LDCEs locate in a CLB. On the left side, we place $LUT4\ LATCH2$, $LUT4\ LATCH1$, $LUT6\ SHIFT1$, $LUT6\ SHIFT2$, $LUT6\ NORTH$, $LUT6\ WEST$, $LUT6\ SOUTH$, and $LUT6\ EAST$ from top to bottom. $LDCE\ SHIFT1$, $LDCE\ SHIFT2$, $LDCE\ NORTH$, $LDCE\ WEST$, $LDCE\ SOURTH$, and $LDCE\ EAST$ locate on the right side. It is important to note that the wires in Figure 7.4 only show the logical connection, which cannot present the correct delay of the post-routing paths. For some critical paths, manual routing is applied for accurate delay control.

7.4.3 FLSH Routing

The most challenging step implementing the FLSH architecture is controlling the wire delay between cells. "Matching delays" is a vain hope in any FPGA, as it is not prac-

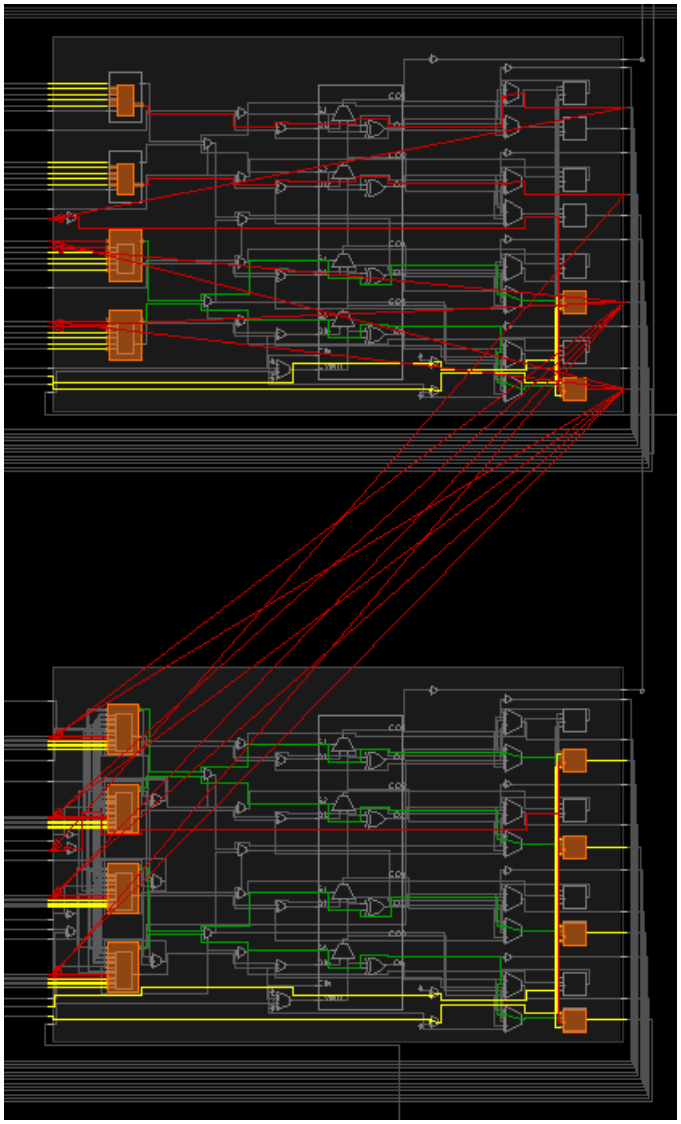

Fig. 7.4: Basic FLSH cell placement

tical. Even though the placement constraints have been applied, Vivado may select unexpected paths, which leads to additional delay. From the statistical results of our implementations, more than 90% of the delay meets the timing constraints to some degree while the rest do not follow the desired routing path. Though we routed the wires

manually to keep the delay equivalent, it is hard to eliminate the difference entirely. As shown in Figure 7.5, the delay between the center cell and neighboring cells is not exactly the same. To minimize the difference, we apply fixed routing constraints. On the other hand, manual routing is known to be very time-consuming for large scale design. By fixing the routing paths, the implementation efficiency can be significantly improved.

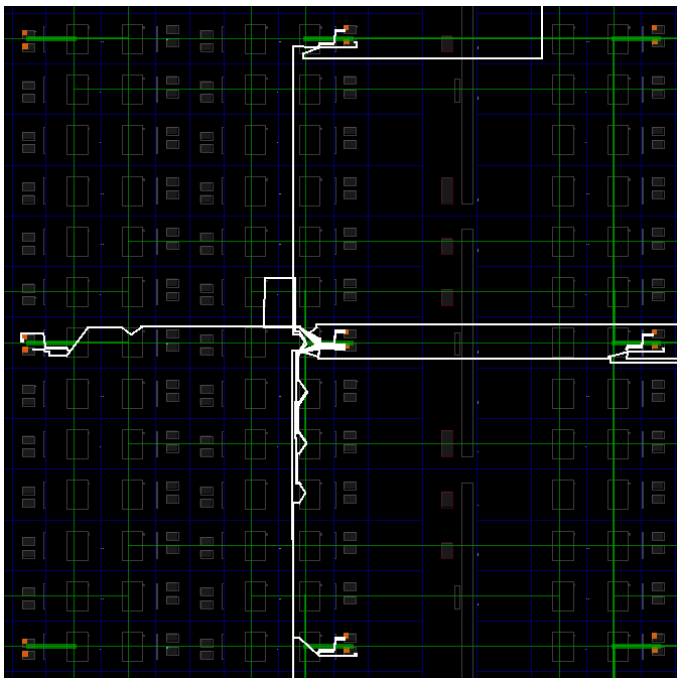

Fig. 7.5: Critical wire routing among cells

7.5 FLSH Application

LSH has been applied to several problem domains including near-duplicate detection, digital fingerprinting, image similarity identification, and hierarchical clustering. In

this section, we will introduce two typical applications of FLSH: similarity search and clustering.

7.5.1 Similarity Search

As a popular similarity search solution, LSH is widely applied for image databases, document collections, time-series databases, and genome databases. Unlike conventional LSHs that can map similar items to the same "buckets" in memory with high probability, FLSH transfers a string to a unique bitmap, as our asynchronous logic in the FPGA holds capabilities to generate similar bitmaps with similar strings. Thus, FLSH can be applied to similarity search application.

The fundamental issue of this application is how to extract the feature from two items efficiently. To post-process the hash value, we provide a simple but practical algorithm called max-min pooling. As shown in Figure 7.6, we first divide the 16×16 cell array into 16 sets. Every set contains 4×4 cells, which means 64 output bits. Then we accumulate the number of '1's in each set. By comparing the sum values, two sets with the maximum and the minimum value are marked as '1' while other sets are '0'. If there are equivalent extremums, we set all of them to '1'. In this way, we compress the 1024-bit hash output to a 4×4 bitmap, which, because of biasing, usually includes 2 '1's and 14 '0's. When we compare the 16-bit sets from similar inputs, they can coincide with high probabilities. Though we use 64-bit input in our example, the actual length can be flexible. As a solution of any shorter input, the missing part is filled with

‘0’. If the length is more than 64-bit, we can divide the input into several 64-bit sections and apply Exclusive or (XOR) operation to fix the length to 64-bit. Alternatively, we use scripts to reconstruct the architecture by extending the scale with more cells.

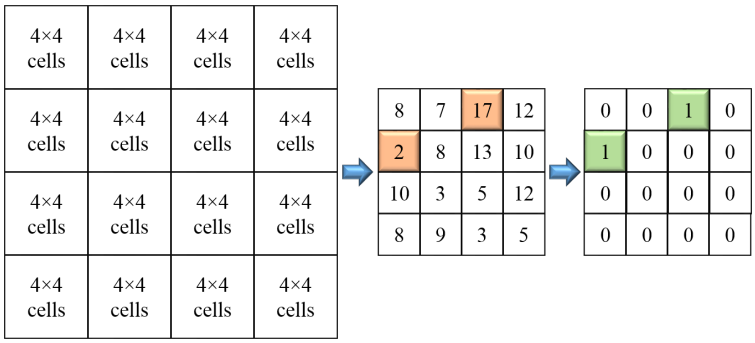

Fig. 7.6: Cell reconstruction and hash regeneration of max-min pooling

7.5.2 Clustering

Clustering is a task of grouping a set of objects so that objects in the same group are more similar than those in other groups. Since FLSH output has the similar feature, we propose a convolutional weight algorithm to develop this application. Figure 7.7 shows a few FLSH cells with different colors. Each cell contains 4 bits. Thus, the total bitmap size is 32×32. In this algorithm, we build a red frame that includes 13 bits. Depending on the location, they are set with different weights. For example, $bit(i, j)$ in the center obtains a weight of 7. Then we give a weight of 3 to $bit(i-1, j)$, $bit(i+1, j)$, $bit(i, j-1)$, $bit(i, j+1)$, which are directly connected to the center bit. Next, a weight of 2 is set to $bit(i-1, j-1)$, $bit(i+1, j-1)$, $bit(i-1, j+1)$, $bit(i+1, j+1)$, which locate in the diagonal direction. Finally, the farthest four bits, $bit(i-2, j)$, $bit(i+2, j)$, $bit(i, j-2)$,

$bit(i, j+2)$, receive the minimum weight of 1. Now we calculate the convolutional weight of $bit(i, j)$ by multiplying every bit with the corresponding weight.

When two items are given for clustering, our FLSH generates the 32×32 bitmap for each input. We shift the center of the red frame from the first $bit(0, 0)$ to the last $bit(31, 31)$ in order to calculate the convolutional weight for each bit. If a bit locates at the boundary of the bitmap, we will fill the missing parts with '0'. After 1,024 shift and calculation, the bitmaps are transferred to two 32×32 "weightmaps". Then we compare the weights in the same location. If the absolute difference is smaller than a pre-defined threshold, the corresponding bits are recorded as matched to a degree. Otherwise, they are not in the same cluster. When most of the weight pairs are matched, these two strings are regarded as similar. The similarity coefficient can be used to evaluate the result of the convolutional weight algorithm for classification tasks. Note that we generate larger hash redundancies than its input to improve the performance. Since the similarity coefficient replaces the bitmap as output, it will not cause more overhead than other LSHs.

7.6 Evaluation and Discussion

In this section, we provide on-board measured results of our FLSH implementation. During the tests, the environment is set to the normal condition. The proposed FLSH needs 2048 LUTs, which takes only 1% resources of the XC7K325T-2FFG900C FPGA. Compared to other LSHs that require processors, the hardware utilization of FLSH is

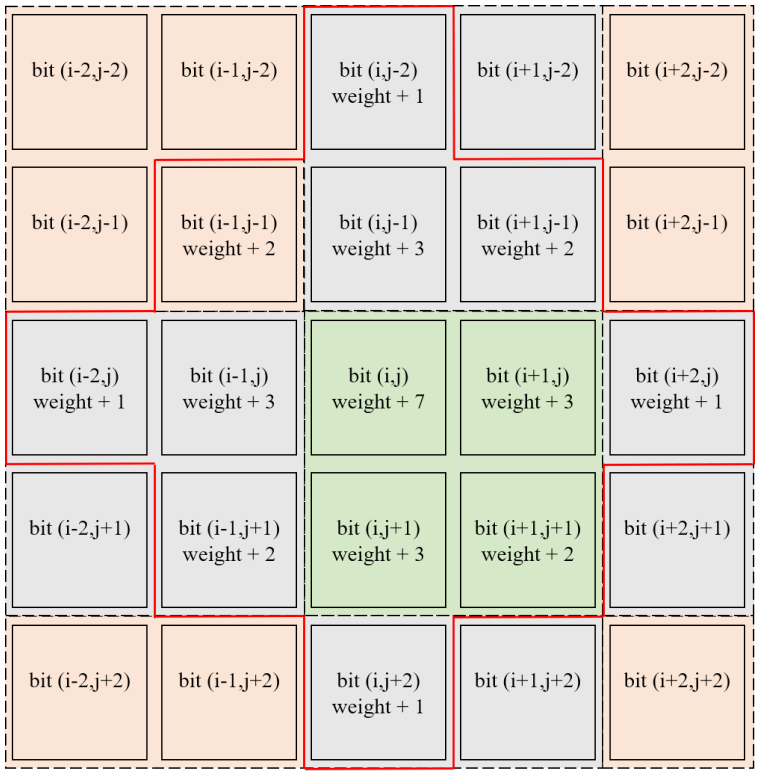

Fig. 7.7: Weight distribution of convolutional weight algorithm

very lightweight.

7.6.1 Convergence Rate

As a hash function, the output is required to be stable. However, FLSHs output random data at the beginning, then it converges gradually. Due to our asynchronous circuit design and wire delay policy, the metastability and randomness are reasonable in the output. To obtain stable hashes, we tried different proportions of the logic LUT initial values. Technically, an obvious bias leads to a faster data convergence rate. However, too many ‘1’s or ‘0’s cause the loss of functionality. We have set LUT initial table

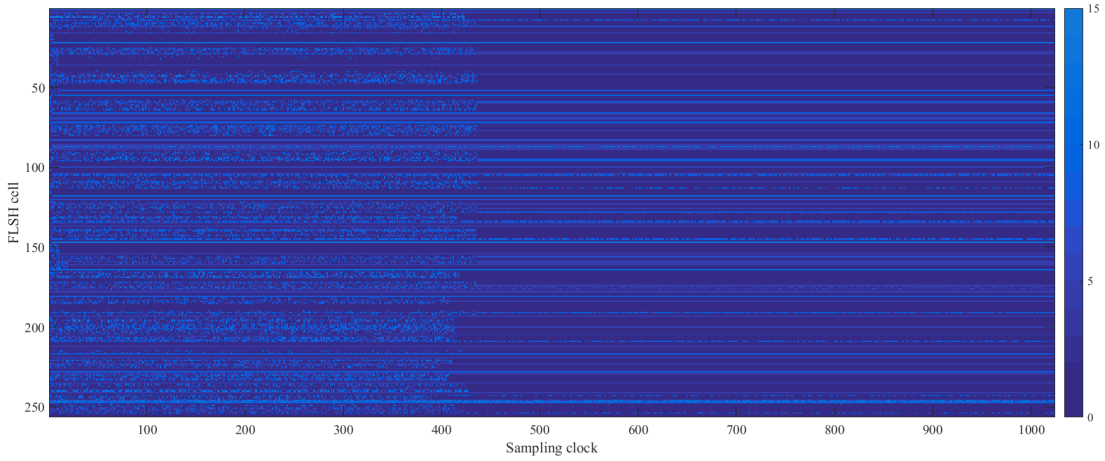

Fig. 7.8: The sample of FLSH output from Xilinx integrated logic analyzer

with 80% of '0' and 20% of '1' as the best tradeoff. To prove that, we used the Xilinx integrated logic analyzer to get 1,024 samples of the FLSH array output, as shown in Figure 7.8. Each column presents the hex values from 256 cells at a certain clock cycle. From left to right, the random hash becomes stable. Particularly, there is an avalanche convergence after the 400th sample. After that, more than 95% of FLSH cells become stable. The time to reach the convergence also decides the speed of FLSH. From our measurement, hash output converges at the 450th sampling clock cycles on average. Since the sampling clock cycle is 5 ns, the speed of FLSHs is calculated to be 444 KHashes/s or 434 Mibps.

Figure 7.9 provides the instability data trends along with the sampling time. If we collect the hash value before the 400th clock cycle, the convergence rate is less than 93%. However, the amount of unstable bits drops suddenly and the curve becomes flat,

as an avalanche convergence occurs in the interval between the 400th and 450th clock cycle. According to our measurements, 450 clock periods are sufficient for the output convergence.

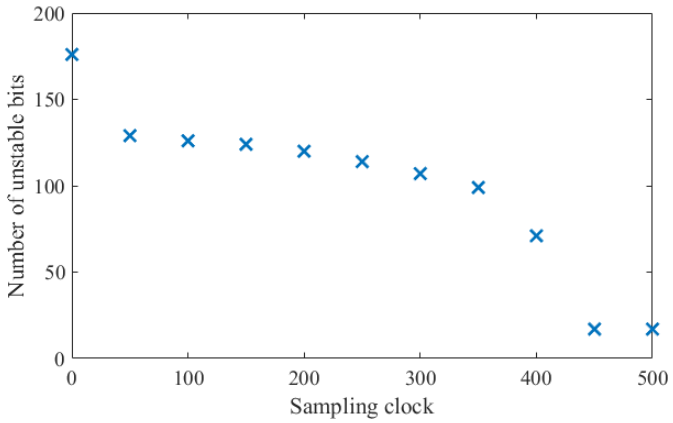

Fig. 7.9: The relationship between the sampling time and the different bits in the output

7.6.2 Randomness

Conventional hash functions have strict randomness requirement for their hash values. Ideally, even if there is only one bit flipped in the input, 50% of the output should change. However, this concept does not fit the LSH family due to its special functionality. LSH output should not be influenced significantly by a few input bits, but it still produces random outputs when 50% of the input bits flip. Therefore, the NIST test [36] is not a suitable standard to evaluate the randomness of LSH. Instead, we show an visual comparison in Figure 7.10. We display bit '1' as white and '0' as black on the 32 $\times$ 32 bitmaps. Given two random inputs, the FLSH generates entirely different hash

values. As we mentioned above, the FLSH output produces more 0 outputs than 1's.

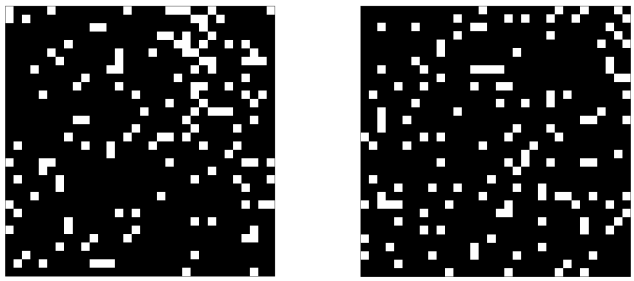

Fig. 7.10: Comparison of 32×32 bitmap with random inputs on an FLSH

Clearly, one sample cannot demonstrate the randomness of FLSHs. To verify it in a statistical sense, we use a linear feedback shift register (LFSR) based pseudo random number generator (PRNG) to generate 10,000 challenges for testing. By accumulating the number of times a '1' appears in each location, we can calculate the probability and evaluate the randomness. From the 32×32 bitmap shown in Figure 7.11, there is no intensive fluctuation on the surface, which means the '1's distribute evenly in the output. Note that the logic LUT input is not always random. It is affected by the shift logic and the initial input values. As a result, the ratio of '1' and '0' in the output may have a bias, compared to the pre-set initial value in Table 7.1.

7.6.3 Fault Tolerant Capability

The fault tolerant capability determines the performance of an LSH. A good LSH should be able to find the similarity between two items without being affected by the existing differences. When the same input is hashed, more '1's are expected to be put

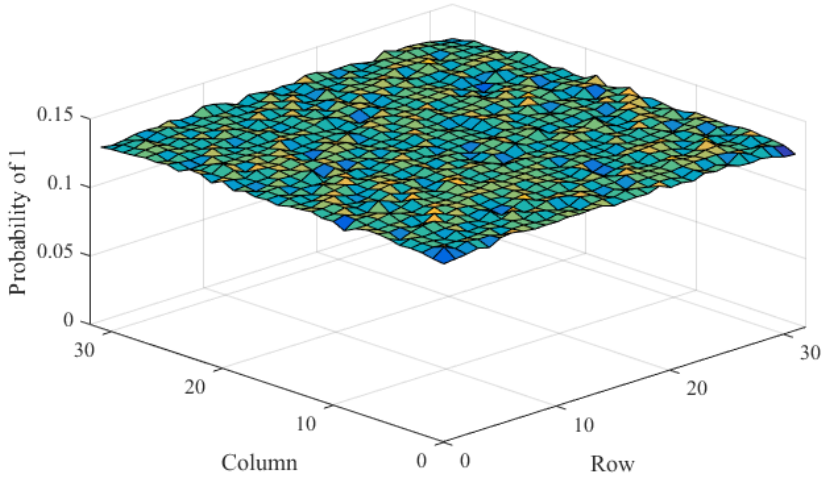

Fig. 7.11: Probability of '1' in the bitmap of FLSH

in the same buckets. Figure 7.12 compares two bitmaps generated by an FLSH with the same input. Since the convergence rate of FLSHs is smaller than 1, some hash values are not stable. As a result, there are 1.7% mismatched bits. However, this defect can be safely ignored as FLSH has strong fault tolerant capability.

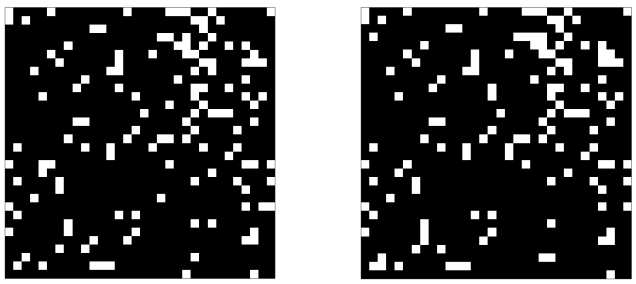

Fig. 7.12: Comparison of 32×32 bitmap with the same input on an FLSH

Figure 7.13 shows the relationship between the flipped bits of the input and the mismatched bits of the relevant output. As increasing errors are injected in the hash

input, FLSH outputs more different bits. When there is more than eight bit errors in the input, the unstable output contains more than 100 bits, which means the stability drops below 90% and may affect the fault tolerant capability. However, this result is acceptable as the raw data of hash output. With our post-processing algorithms, we are still able to improve the performance of the FLSH-based applications.

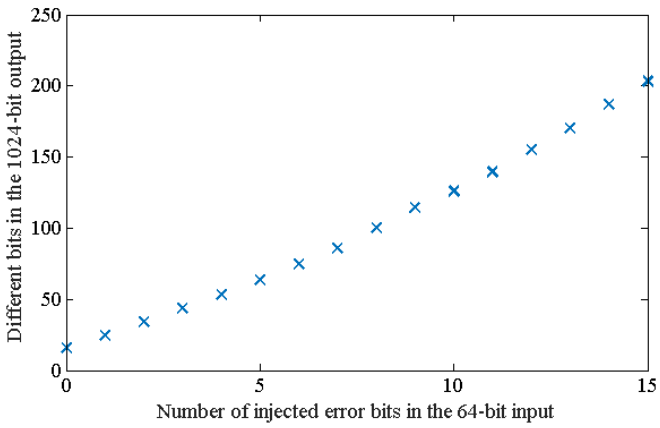

Fig. 7.13: FLSH output instability with different input errors

7.6.4 Similarity Search Result

The performance of FLSH-based similarity search is shown in Table 7.2. As a comparison with the traditional LSHs, the Hamming and Minhash similarity rate are tested with the same data set. Though the max-min pooling output is compressed to only 16 bits, the algorithm still provides a good similarity rate as the Hamming similarity does. Thus, given the 64-bit strings in our example, max-min pooling algorithm can be safely applied to similarity search.

Table 7.2: Performance of FLSH-based Similarity Search

different bits in two FLSH 64-bit input	4	5	6	7	8	9	10
Hamming similarity of 64-bit input	93.75%	92.19%	90.63%	89.06%	87.50%	85.94%	84.38%
Jaccard similarity of Minhash	88.24%	85.51%	82.86%	80.28%	77.78%	75.34%	73.33%
similarity rate of max-min pooling algorithm	92.06%	91.22%	90.35%	89.87%	88.01%	85.72%	82.90%

Apart from the ability to detect similarity, we also need to show that FLSH can discriminate between dissimilar inputs. We use PRNG to generate two input data sets, with 10,000 random strings in each group. According to the results of the FLSH and convolutional weight algorithm, there are 437 mismatched bits in the 1,024-bit output on average. If we set the threshold to 100, the application can distinguish two random strings with a probability of almost 100%.

7.6.5 Clustering Result

According to Table 7.3, the number of mismatched bits in the bitmap increases when there are more different bits in the input sets. The same data shown in Figure 7.13 demonstrates the relationship between input bit errors and output bit errors. However,

the amount of mismatched bits grows faster after the fifth column. As a result, the Hamming similarity of FLSH bitmaps drops rapidly. When we apply the convolutional weight algorithm, the mismatched bits are significantly reduced. We set the weight threshold to 8 in our tests. Even though there are 10 different bits in the 64-bit input, our solution can generate 1,012 matched bits. The similarity rate is improved to 98.83%, which means only 1.17% inputs cannot be classified correctly. While the max-min pooling algorithm fits similarity search well, the convolutional weight algorithm performs better for clustering application.

Table 7.3: Performance of FLSH-based Clustering

different bits in two 64-bit input sets	4	6	8	10	12	14	16
mismatched bits among FLSH bitmaps	53	75	100	126	155	190	227
Hamming similarity of FLSH bitmaps	94.82%	92.68%	90.23%	87.70%	84.86%	81.44%	77.83%
matched bits of convolutional weight algorithm	1024	1021	1018	1012	996	960	909
similarity rate of convolutional weight algorithm	100.00%	99.71%	99.41%	98.83%	97.27%	93.75%	88.77%

As was noted earlier, with a 200 MHz clock on the FPGA and a convergence time of 450 samples, the FLSH can produce over 444 KHashes/s. The convolutional weight algorithm can be pipelined with the FLSH process, thus keeping the same hash rate of 444 KHashes/s. As a point of comparison, a Python software implementation of Minhash delivers only 3279 hashes/s on an Intel i7-4770 processor, roughly 135 times slower than the FPGA-based FLSH implementation.

7.7 Conclusions

In this chapter, an FPGA-based locality sensitive hash function is proposed to identify similar strings. We present the details of FLSH architecture, implementation, convergence, randomness, and stability. When there are bit errors in the hash input, our FLSH can tolerate them and keep most output unchanged. In the similarity search and clustering application, our algorithms provide reasonable performance and improvement. In summary, this FPGA based locality sensitive hash function is an efficient solution for identification and classification.

Chapter 8

Conclusions

Though many security solutions have been developed recently for the increasing needs of embedded systems and IoTs, there is still space for further improvement in the performance, cost, and security level of these systems. The trade-off becomes significant, since quite a few devices can not support very strong encryption with large computational complexity, but still require a certain degree of security. Therefore, designers must consider all aspects and find a balance between performance and cost. Physical unclonable functions, thus, become an ideal security primitive with strong advantages. In this thesis, we developed new architectures and novel techniques to overcome some of the main shortcomings of PUFs. The major contributions of our work will be summarized in this chapter.

8.1 PUF Design Optimization and Implementation

Physical unclonable functions are known as a security primitive for various applications. Its instability limits the usage as a strong error correction code is usually required,

which increases the total hardware resource utilization significantly. In chapter 2, several optimizations on the hardware architecture are proposed in order to improve the stability and reduce the system cost. With a shift register or hash function implementation on the FPGA, the required number of ROs decreases to only 128. By involving the IDELAY2 module, we are able to adjust the timing of the RO output and improve the stability.

8.2 Secure Authentication Solution with PUF Response Instability

For most PUF research, unstable bits generated by RO pairs are usually regarded as a negative part that should be eliminated or ignored for output. However, we demonstrate its positive aspect in chapter 3. By converting the unstable bits to stable output, using unstable-stable responses is proved to be a practical solution for authentication. Compared to the traditional ECC methods, USR achieves better performance. We also propose another algorithm using the instability of PUFs. The obfuscation design can guarantee a high security level of PUF responses against modeling attacks.

8.3 PUF Initialization Table for Robust Authentication and Key Generation

For some applications that need high accuracy, the bit error rate in PUF responses should be as low as possible. In chapter 4, we comprehensively evaluate the lower bound of PUF BER on FPGAs. We propose a floating thresholding solution to reduce the bit error rate in PUF responses and improve the performance of PUF-based appli-

cations. We combine the error correction scheme and bit selection scheme together by using an initialization table and unique helper data. Our results show that this novel approach improves the stability and data rate of various PUFs significantly.

8.4 Ring Weight Algorithm for Lightweight Authentication

Counterfeit Integrated Circuits (IC) can be very harmful to the security and reliability of critical applications. Physical Unclonable Functions (PUF) have been proposed as a mechanism for uniquely identifying ICs and thus reducing the prevalence of counterfeits. However, maintaining large databases of PUF challenge response pairs (CRPs) and dealing with PUF errors make it difficult to use PUFs reliably. To satisfy the authentication requirement of low-cost devices, we proposed a novel fuzzy identification scheme in chapter 5. It presents an innovative approach to authenticate CRPs on PUF-based ICs. The proposed method can tolerate considerable bit errors from responses of PUFs without the use of error correcting codes. Different types of optimization methods are applied to improve the overall performance.

8.5 Locality Sensitive Hash Function for Similarity Search and Clustering

Locality sensitive hash functions (LSHs) are a widely used approach to identify similar items or nearest neighbors in a data set. However, LSH algorithms have rarely been explored in hardware implementations. In chapter 6, we propose a novel locality sensitive hash function design and implementation using FPGAs. By using the unique

hardware delay generated during the fabrication process, we can identify the bitmaps from similar inputs. Our results have verified the efficiency and accuracy of our LSH for similarity search and clustering applications.

8.6 Summary of Conclusions

This thesis is devoted to a PUF-based trade-off design and algorithms for authentication and key generation applications on embedded systems and IoTs.

Bibliography

[1] G.E. Suh and S Devadas. Physical unclonable functions for device authentication and secret key generation. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 9–14, June 2007.

[2] Jorge Guajardo, SandeepS. Kumar, Geert-Jan Schrijen, and Pim Tuyls. FPGA intrinsic PUFs and their use for IP protection. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 63–80. Springer Berlin Heidelberg, 2007.

[3] Blaise Gassend. *Physical Random Functions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, 2003.

[4] C. Herder, Meng-Day Yu, F. Koushanfar, and S. Devadas. Physical unclonable functions and applications: A tutorial. *Proceedings of the IEEE*, 102(8):1126–1141, Aug 2014.

[5] Jeroen Delvaux, Dawu Gu, Dries Schellekens, and Ingrid Verbauwhede. Helper data algorithms for PUF-based key generation: Overview and analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34 (6):889–902, 2015.

[6] Meng-Day Yu and Srinivas Devadas. Secure and robust error correction for physical unclonable functions. *IEEE Design & Test of Computers*, 27(1):48–65, 2010.

[7] Bohdan Karpinskyy, Yongki Lee, Yunhyeok Choi, Yongsoo Kim, Mijung Noh, and Sanghyun Lee. 8.7 physically unclonable function for secure key generation with a key error rate of 2e-38 in 45nm smart-card chips. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 158–160. IEEE, 2016.

[8] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. Low-overhead implementation of a soft decision helper data algorithm for sram pufs. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 332–347. Springer, 2009.

[9] Charles Herder, Ling Ren, Marten van Dijk, Meng-Day Yu, and Srinivas Devadas. Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions. *IEEE Transactions on Dependable and Secure Computing*, 14(1):65–82, 2017.

[10] Md Tauhidur Rahman, Domenic Forte, Fahim Rahman, and Mark Tehranipoor. A pair selection algorithm for robust ro-puf against environmental variations and aging. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, pages 415–418. IEEE, 2015.

[11] Abhranil Maiti, Jeff Casarona, Luke McHale, and Patrick Schaumont. A large scale characterization of RO-PUF. In *Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on*, pages 94–99. IEEE, 2010.

[12] Dominik Merli, Frederic Stumpf, and Claudia Eckert. Improving the quality of ring oscillator PUFs on FPGAs. In *Proceedings of the 5th Workshop on Embedded Systems Security*, page 9. ACM, 2010.

[13] Haile Yu, Philip Heng Wai Leong, Heiko Hinkelmann, L Moller, Manfred Glesner, and Peter Zipf. Towards a unique FPGA-based identification circuit using process variations. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 397–402. IEEE, 2009.

[14] Abhranil Maiti and Patrick Schaumont. Improving the quality of a physical unclonable function using configurable ring oscillators. In *2009 International Conference on Field Programmable Logic and Applications*, pages 703–707. IEEE, 2009.

[15] Chi-En Yin, Gang Qu, and Qiang Zhou. Design and implementation of a group-based ro puf. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 416–421. EDA Consortium, 2013.

[16] Yohei Hori, Takahiro Yoshida, Toshihiro Katashita, and Akashi Satoh. Quantitative and statistical performance evaluation of arbiter physical unclonable functions on fpgas. In *Reconfigurable Computing and FPGAs (ReConFig), 2010 International Conference on*, pages 298–303. IEEE, 2010.

[17] Ulrich Rührmair, Jan Sölter, Frank Sehnke, Xiaolin Xu, Ahmed Mahmoud, Vera Stoyanova, Gideon Dror, Jürgen Schmidhuber, Wayne Burleson, and Srinivas Devadas. Puf modeling attacks on simulated and silicon data. *IEEE Transactions on Information Forensics and Security*, 8(11):1876–1891, 2013.

[18] Gabriel Hospodar, Roel Maes, and Ingrid Verbauwhede. Machine learning attacks on 65nm arbiter pufs: Accurate modeling poses strict bounds on usability. In *WIFS*, pages 37–42, 2012.

[19] Qingqing Chen, György Csaba, Paolo Lugli, Ulf Schlichtmann, and Ulrich Rührmair. The bistable ring PUF: A new architecture for strong physical unclonable functions. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 134–141. IEEE, 2011.

[20] Sandeep S Kumar, Jorge Guajardo, Roel Maes, Geert-Jan Schrijen, and Pim Tuyls. The butterfly PUF protecting IP on every FPGA. In *Hardware-Oriented Security and Trust, 2008. HOST 2008. IEEE International Workshop on*, pages 67–70. IEEE, 2008.

[21] C. Keller, F. Gurkaynak, H. Kaeslin, and N. Felber. Dynamic memory-based physically unclonable function for the generation of unique identifiers and true random numbers. In *Circuits and Systems (ISCAS), 2014 IEEE International Symposium on*, pages 2740–2743, June 2014.

[22] Fatemeh Tehranipoor, Nima Karimian, Wei Yan, and John A Chandy. Dram-based intrinsic physically unclonable functions for system-level security and authentication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2016.

[23] M. Cortez, A Dargar, S. Hamdioui, and G.-J. Schrijen. Modeling sram start-up behavior for physical unclonable functions. In *Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2012 IEEE International Symposium on*, pages 1–6, Oct 2012.

[24] Mohammad Tehranipoor and Cliff Wang. *Introduction to hardware security and trust*. Springer Science & Business Media, 2011.

[25] Lilian Bossuet, Xuan Thuy Ngo, Zouha Cherif, and Viktor Fischer. A PUF based on a transient effect ring oscillator and insensitive to locking phenomenon. *IEEE Transactions on Emerging Topics in Computing*, 2(1):30–36, 2014.

[26] *7 Series FPGAs Configuration User Guide*. Xilinx, 9 2016. Rev. 1.

[27] *Frequency Counter for Spartan-3E Starter Kit*. Xilinx, 3 2006. Rev. 2014.1.

[28] Takanori Machida, Dai Yamamoto, Mitsugu Iwamoto, and Kazuo Sakiyama. A new arbiter PUF for enhancing unpredictability on FPGA. *The Scientific World Journal*, 2015.

[29] Alexander Wild, Georg T Becker, and Tim Güneysu. On the problems of realizing reliable and efficient ring oscillator PUFs on FPGAs. In *Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on*, pages 103–108. IEEE, 2016.

[30] *Vivado Design Suite User Guide*. Xilinx, 2 2014. Rev. 1.

[31] *7 Series FPGAs Configurable Logic Block User Guide*. Xilinx, 11 2014. Rev. 1.7.

[32] *7 Series FPGAs SelectIO Resources User Guide*. Xilinx, 9 2015. Rev. 1.6.

[33] Xilinx. Xilinx Kintex-7 IBIS. https://www.xilinx.com/content/xilinx/en/-downloadNav/device-models/ibis-models/kintex-series-fpgas.html. Accessed: 2013-09-23.

[34] *Kintex-7 FPGAs Data Sheet: DC and AC Switching Characteristics*. Xilinx, 11 2015. Rev. 2.15.

[35] A Maiti, L. McDougall, and P. Schaumont. The impact of aging on an FPGA-based physical unclonable function. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 151–156, Sept 2011.

[36] *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*. National Institute of Standards and Technology, 4 2010. Rev. 1a.

[37] Wei Yan, Fatemeh Tehranipoor, and John A Chandy. A novel way to authenticate untrusted integrated circuits. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pages 132–138. IEEE Press, 2015.

[38] Ujjwal Guin, Domenic Forte, and Mohammad Tehranipoor. Anti-counterfeit techniques: From design to resign. In *Proc. International Workshop Microprocessor Test and Verification*, pages 89–94, 2013.

[39] Blaise Gassend, Dwaine Clarke, Marten van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 148–160, New York, NY, USA, 2002. ACM.

[40] Christoph Bösch, Jorge Guajardo, Ahmad-Reza Sadeghi, Jamshid Shokrollahi, and Pim Tuyls. Efficient helper data key extractor on FPGAs. In *Cryptographic Hardware and Embedded Systems–CHES 2008*, pages 181–197. Springer, 2008.

[41] Daisuke Suzuki and Koichi Shimizu. The glitch PUF: A new delay-PUF architecture exploiting glitch shapes. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 366–382. Springer Berlin Heidelberg, 2010.

[42] Fatemeh Tehranipoor, Nima Karimian, Kan Xiao, and John Chandy. DRAM based intrinsic physical unclonable functions for system level security. In *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 15–20, New York, NY, USA, 2015. ACM.

[43] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[44] Myeong-Hyeon Lee and Yoon-Hwa Choi. A fault-tolerant Bloom filter for deep packet inspection. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 389–396, Dec 2007.

[45] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better Bloom filter. In Yossi Azar and Thomas Erlebach, editors, *Algorithms, ESA 2006*, volume 4168 of *Lecture Notes in Computer Science*, pages 456–467. Springer Berlin Heidelberg, 2006.

[46] Ryo Nojima and Youki Kadobayashi. Cryptographically secure Bloom-filters. *Transactions on Data Privacy*, 2(2):131–139, 2009.

[47] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[48] Meng-Day Yu and S. Devadas. Secure and robust error correction for physical unclonable functions. *Design Test of Computers, IEEE*, 27(1):48–65, Jan 2010.

[49] Wenjie Che, Fareena Saqib, and Jim Plusquellic. PUF-based authentication. In *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*, pages 337–344. IEEE, 2015.

[50] Andrew Rukhin, Juan Soto, James Nechvatal, Elaine Barker, Stefan Leigh, Mark Levenson, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report Special Publication 800-22, NIST, 2010.

[51] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of Massive Datasets*, volume 1. Cambridge University Press Cambridge, 2012.

[52] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[53] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, volume 99, pages 518–529, 1999.

[54] Ondrej Chum, James Philbin, Andrew Zisserman, et al. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC*, volume 810, pages 812–815, 2008.

[55] Jonathan Oliver, Chun Cheng, and Yanggui Chen. Tlsh–a locality sensitive hash. In *Cybercrime and Trustworthy Computing Workshop (CTC), 2013 Fourth*, pages 7–13. IEEE, 2013.